6-1-2000

A VHDL design for hardware assistance of fractal image compression

Andrew Erickson

A VHDL Design for Hardware Assistance of Fractal Image Compression

by

Andrew Erickson

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in
Computer Engineering

Approved by:

Principle Advisor _____
Muhammad E. Shaaban, Assistant Professor, Computer Engineering

Committee Member _____
Kenneth W. Hsu, Professor, Computer Engineering

Committee Member _____
Roy S. Czernikowski, Professor, Computer Engineering

Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2000

Release Permission Form

Rochester Institute of Technology

A VHDL Design for Hardware Assistance of Fractal Image Compression

I, Andrew J. Erickson, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

Andrew J. Erickson

26 June 2000

Date

Abstract

Fractal image compression schemes have several unusual and useful attributes, including resolution independence, high compression ratios, good image quality, and rapid decompression. Despite this, one major difficulty has prevented their widespread adoption: the extremely high computational complexity of compression.

Fractal image compression algorithms represent an image as a series of contractive transformations, each of which maps a large domain block to a smaller range block. Given only this set of transformations, it is possible to reconstruct an approximation of the original image by iteratively applying the transformations to an arbitrary image.

Compression consists of partitioning the image into range blocks and finding a suitable transformation of a domain block to represent each one. This search for transformations must generally be done using a brute force approach, comparing successive domain blocks until a suitable match is found. Some algorithmic improvements have been found, but none are adequate to reduce the required compression time to something reasonable for many uses.

This thesis presents a new ASIC design which performs a large number of the required comparisons in parallel, yielding a substantial speedup over a program on a general-purpose computer system. This ASIC is designed in VHDL, which may be synthesized to many different target architectures. The design has considerable flexibility which makes it applicable to different images and applications.

The design is based around a pipeline of units that each compare one range block with a series of domain blocks which are fed through the pipeline. Comparisons are made to minimize the mean square error (MSE) of a transform given a linear mapping of the intensity values. This is, by far, the most common minimization strategy used in the literature.

The speedup provided by this design is estimated to be about 1,000 times for 256×256 images divided into 8×8 blocks over a sequential processor given similar implementation technologies.

Contents

List of Figures

List of Tables

Glossary

Page numbers in brackets refer to locations in the text where the term defined is discussed in greater detail. Terms which have no such references are merely mentioned in passing in the text.

affine transformation A transformation on a matrix of the form $T(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$, where $\mathbf{x}$ and $\mathbf{b}$ are column vectors and $\mathbf{A}$ is a square matrix.

ASIC Application Specific Integrated Circuit; a complex digital circuit which performs some specialized function (as opposed to a general-purpose device).

attractor [4] The limiting value of a contractive transformation. The attractor may be approximated by repeatedly applying the transformation to a seed value.

chunk [17] In the design developed here, a chunk is a subcomponent within a unit.

complete [3] A metric space is complete if there is always a point between any two given points (if it has no "holes").

contraction mapping theorem [4] A key theorem for fractal image compression; it states that a contractive transform on a metric space has a single fixed point attractor.

contractive [3] A transformation on a space is contractive if its contractivity factor is strictly less than one.

contractivity factor [3] A (fractional) upper limit on the change in the distance between any two points after a transformation is applied to them.

Design Compiler A VHDL synthesis tool which is part of Synopsys.

domain block [10] A block in an image which is the source of a transformation.

entropy encoding Any general-purpose (lossless) compression scheme which relies on disparities in the relative frequency of various bit patterns to compress data. Entropy encoding of some sort is often used as a last step in fractal image compression.

FIFO [54] First In First Out; a cache eviction strategy where the item which has been in the cache the longest is evicted when a new item is inserted, regardless of intervening accesses.

fixed point [4] A point in a transformation on a space which remains unchanged by the transformation; in other words, $x_f = T(x_f)$.

fractal A mathematical set which exhibits self-similarity. Typically, such sets are represented graphically.

fractal image compression [10] A class of algorithms for image compression which are based on fractal theory; they are generally based on LIFSMs.

fractal transform [7] Another name for an LIFSM sometimes encountered in the literature.

gzip A popular lossless compression system which uses Lempel-Ziv (LZ77) coding.

$\mathcal{H}$ [5] The Hausdorff set.

Hausdorff metric [5] A metric which defines the distance between two subsets of a metric space based on the distance metric of the underlying space. With the Hausdorff set, it forms a metric space.

Hausdorff set [5] The set of all possible subsets of a space except the empty set.

HDL Hardware Description Language; any of several programming languages used for describing digital hardware operation, generally for simulation, synthesis, or human communication.

IFS [5] Iterated Function System; a set of transformations operating on a metric space which defines a fractal.

IFSM [6] Iterated Function System with Graymap Functions; an IFS with an additional dimension for intensity information.

JPEG Joint Picture Experts Group; a research group known for a common image compression algorithm based on the discrete cosine transform. Also, this compression algorithm.

LIFS [6] Local Iterated Function System; an IFS in which each transformation acts on a subset of the space, rather than the whole.

LIFSM [7] Local Iterated Function System with Graymap Functions; a LIFS with an additional dimension for intensity information. An image compressed with fractal image compression is generally an LIFSM.

lossless A compression algorithm which reproduces its input precisely after decompression.

lossy A compression algorithm which is not guaranteed to reproduce the precise original after decompression.

metric [3] A distance measure between points in a space.

metric space [3] A space and an associated metric.

MSE [28] Mean Square Error; a common distance function and error measure.

PIFS [6] Partitioned Iterated Function System; another name for an LIFS sometimes found in the literature.

QD [12] Quadtree Decomposition.

QR [13] Quadtree Recomposition.

quadtree A tree structure where each non-leaf node has (at most) four children. In fractal image compression, a quadtree is formed when a square range block is split into four smaller range blocks by bisecting the edges.

quadtree decomposition [12] The partitioning of an image into square blocks of varying size using a quadtree approach, where large blocks are examined and are recursively split into four equal smaller blocks as needed.

quadtree recomposition [13] An algorithmic variant of quadtree decomposition where the tree structure is constructed starting with the smallest blocks and joining them as appropriate.

RAM Random Access Memory; a term which is universally applied solely to semiconductor memories which allow both reading and writing.

range block [10] A block in a LIFS which is the target of a transformation. When compressing an image, it is completely divided into non-overlapping range blocks.

s [3] The contractivity factor.

space [3] An infinite set upon which topological transformations are performed. (In this thesis, all spaces considered are metric spaces.)

Synopsys A well-known digital logic development system; the Synopsys Design Compiler was used to synthesize the design presented here.

synthesis [50] The process, generally performed by a computer system, of automatically generating a logic design from an HDL description.

transformation [3] A function which maps a member of a space onto another member of the same space. In fractal image compression, this is generally limited to a subset of the affine transforms.

unit [17] In the design developed here, the primary components are referred to as units. Units may consist of smaller subcomponents named chunks.

VHDL VHSIC Hardware Description Language; a popular hardware description language developed by the Department of Defense. VHDL was developed to support simulation, but is now also frequently used for synthesis.

VHSIC Very High Speed Integrated Circuit; a Department of Defense program which, among other things, developed VHDL.

VQ [8] Vector Quantization; a class of block-based image compression algorithms where blocks are assumed to take the form of one of a number of possible parameterized blocks in a fixed codebook; the block indices and parameters are encoded.

Chapter 1

Introduction

Fractal image compression has several features which would seemingly make it ideal for many applications, including resolution independence, rapid decompression, and high compression ratios. Despite this, it has remained of only secondary importance, primarily because of one difficulty—the very high computational cost of compression. Despite ongoing research, software improvements which reduce the time required for compression enough to be practical for general purpose use have not been forthcoming.

The algorithms required for fractal image compression are easily performed in parallel; the complexity of the software is due to a very large number of relatively simple cases. This makes fractal image compression an attractive problem to perform in special-purpose hardware.

1.1 Scope of Work

In this thesis, a new design for an ASIC to assist with fractal image compression is presented. This ASIC is designed using VHDL, which allows flexibility in implementation; it is not tied to any particular system architecture.

The design presented is based around a pipeline of many identical units which perform the block comparisons necessary for fractal image compression in parallel. This allows a substantial speedup when compared with software written for general purpose processors.

The system was tested using a VHDL simulation, and the results compared to that obtained from compression software. The design was also synthesized to a gate level using Synopsys synthesis tools. Although no physical devices were actually constructed to test the design, the testing which was done should ensure that such devices do work as expected.

1.2 Organization of the Material

The remainder of this thesis covers three major topics. First, some background on fractal image compression is given. Chapter 2 discusses the mathematical theory

supporting fractal image compression—the theories of local iterated function systems. It also contains a brief summary of some alternate theoretical approaches to fractal image compression. Chapter 3 describes the algorithm generally used for fractal image compression, and briefly describes some of the improvements on this general algorithm. This chapter also describes some related hardware designs which have been proposed.

The second section describes, in detail, the ASIC design developed for this thesis. Chapter 4 gives an overview of the design; chapters 5, 6, and 7 each describe one of the three major components of the design.

The third section considers the testing and analysis of this design. Chapter 8 describes the testing of the design by verifying the expected operation of the ASIC and by checking the underlying algorithm. Chapter 9 describes the results of synthesizing the design using Synopsys, and chapter 10 contains some concluding observations and some suggestions for future work on this design.

Appendices contain source code for a software fractal image compression and decompression system, a copy of the script files used to synthesize the design, a representative selection of schematics generated from the synthesis, and a brief description of some utility programs developed in conjunction with this thesis. The accompanying CD-ROM contains source code to these utility programs, along with the VHDL code for the ASIC design and an electronic copy of this thesis.

Chapter 2

Fractal Theory and Image Compression

2.1 Images and Transformations

An image consists of a metric space with an associated set of chromatic attributes. A metric space (X, d) is composed of a topological space X (e.g. an infinite set) and a metric d—a measure of distance. A metric is a non-negative real valued function of two points in the space which obeys three requirements [2, 11]:

1. $d(x, y) = 0$ iff $x = y$
2. $d(x, y) = d(y, x)$ (reflexivity)
3. $d(x, y) \leq d(x, z) + d(z, y)$ (the triangular inequality)

For images, the space used is a plane, and the distance measure is commonly the Euclidean distance (or Euclidean metric). Other metric spaces are used from time to time with fractal manipulations.

A metric space (X, d) is complete if, for any points $x, y \in X$ for which $d(x, y) > 0$, there exists a point z for which $d(x, z) < d(x, y)$ and $d(y, z) < d(x, y)$. In other words, complete metric spaces are continuous.

The chromatic attributes of an image are assumed throughout this document to consist of a numerical equivalent of an intensity—a grayscale value. Actual real-world images contain color information at a variety of wavelengths, which may be represented in a computer with any of several color models, such as the RGB model. The algorithms described herein may be applied to each color component individually if non-grayscale images are used.

A transformation is a mapping of a space onto itself (a function of points in the space yielding points in the space). A transformation T is said to be contractive with contractivity factor s if

$$d(T(x), T(y)) \leq sd(x, y), 0 \leq s < 1$$

That is, the distance between an arbitrary pair of points becomes less by a factor of at least s after applying the transformation. For fractal image compression, the transformations are usually restricted to a subset of contractive affine transformations, although the theories upon which fractal image compression is based hold for any contractive transformations.

2.1.1 Contraction Mapping Theorem

Every contractive transformation on a complete metric space has an attractor which is invariant across the transformation. A sequence formed by repeatedly applying the transformation to any subset of the space has, as its limiting value, this fixed point. This is the Contraction Mapping Theorem.

Theorem 1 (Contraction Mapping Theorem) *A contraction mapping T on a complete metric space (X, d) has exactly one fixed point $x_f \in X$ (for which $T(x_f) = x_f$) and*

$$\lim_{n\to\infty} T^{\circ n}(x) = x_f.$$

where $T^{\circ 1}(x) = T(x)$, and $T^{\circ n}(x) = T[T^{\circ(n-1)}(x)]$ for $n > 1$.

Proof *Let $x_1, x_2, \ldots, x_n \in X$. Since T is contractive,*

$$\max\{d(x_a, x_b)\} \geq s \cdot \max\{d(T(x_a), T(x_b)\}$$

for $a, b \leq n$. Let $\epsilon > 0$ be given. Then,

$$\max\{d(T^{\circ n}(x_a), T^{\circ n}(x_b)\} \leq s^n \max\{d(x_a, x_b)\} \leq \epsilon \text{ if } n \geq \frac{s \cdot \max\{d(x_a, x_b)\}}{\epsilon}.$$

This proves there is a single attractor for any set of points in the space (and, by extension, for all points in the space). Further, as the space is complete, the limiting value x_f is a member of X. x_f is fixed because $d(x_f, x_f) = 0$ and thus $d(x_f, T(x_f)) = 0$ (as x_f is the attractor.)

This is proven, with similar reasoning but different notation, in [1, 72–73]. This proof also implies that the error (distance between the attractor and a sequence value) is constrained by a monotonically decreasing upper bound as the number of iterations increases. Thus, the limit of the sequence may be approximated to within any desired error bound by making a finite number of transformations.

2.2 Fractals

A fractal is a set (often represented visually) which exhibits self-similarity. Fractals are, precisely speaking, defined only for a metric space and do not directly involve any chromatic information associated with it. (An alternate approach is to consider the chromatic information to lie along an additional dimension of the

space and to define a new metric which includes this dimension. This approach is useful in developing the theory but is not frequently used in practice.)

Many mathematical constructs lead to fractals; for the purpose of image compression, however, fractals based on local iterated function systems (LIFSs) are used exclusively. A LIFS is a modification of an iterated function system (IFS).

2.2.1 Iterated Function Systems

An iterated function system consists of a set T of contractive transformations on a metric space A with a distance metric d_A. These transformations act in parallel—the union of all of the transformations is iterated.

For analysis, a new metric space is constructed, called the Hausdorff Space $\mathcal{H}_A$. $\mathcal{H}_A$ is defined on the space of all subsets of A (the power set of A) except the empty set, with a metric $d_{\mathcal{H}}$ defined below. (Intuitively, this definition is essentially the "maximum overlap" between the sets.)

Let $a, a_1, a_2, \ldots \in A$ and $A, A_1, A_2, \ldots \in \mathcal{H}_A$. Then, one may define:

$$\begin{aligned} d(a, A) &= \min\{d_a(a, a_1) : a_1 \in A\} \\ d(A_1, A_2) &= \max\{d(a_1, A_2) : a_1 \in A_1\} \\ d_{\mathcal{H}}(A_1, A_2) &= \max\{d(A_1, A_2), d(A_2, A_1)\} \end{aligned}$$

$d_{\mathcal{H}}$ is the Hausdorff distance.

An IFS may be characterized by a single contractive transformation in $\mathcal{H}_A$. A single contractive transformation on a metric space is also a contractive transformation on the associated Hausdorff space with the Hausdorff metric. (Since the transformation maps any subset of the original space onto another subset, it clearly is a transformation of points in the Hausdorff space. It is contractive by nature of the definition of $d_{\mathcal{H}}$, which is the distance of some particular pair of points in the original metric space. This distance must be smaller by at least a factor of s after the transformation; hence, $d_{\mathcal{H}}$ must also be smaller by at least the same factor.)

Similar reasoning applies to a set of contractive transformations; they (the union of the transformations) correspond to a single contractive transformation in the Hausdorff metric space with a contractivity factor of the maximum of the contractivity factors of the component transformations. This implies, by the Contraction Mapping Theorem 1, that there exists a single fixed point in $\mathcal{H}$ which is the attractor for the IFS. A point in $\mathcal{H}$ is equivalent to a subset of the original metric space. Further, a finite number of applications of the IFS to any point in $\mathcal{H}$ will suffice to approximate, to within a desired error, this attractor.

For grayscale images, one may use a three-dimensional metric space for the IFS, with the third dimension representing the intensity of a point. This may lead to, in the fixed attractor, one location in the image having several simultaneous intensity values; for display, these values are of necessity combined into a single

value. This is commonly done by either taking the maximum value or the mean value of the set of intensities, although other algorithms are possible.

IFSs which are applied to grayscale images in this way (treating the intensity as a third axis) are often IFSMs (Iterated Function Systems with Graymap functions), where the graymap function transforms the intensity at a point in the metric space independently of the location transformation. Although the transformations are usually handled separately in practice, the theory is simpler and more general without this requirement.

In practice, the combining of multiple grayscale values takes place at each iteration, as the storage requirements for a two-dimensional structure are less than for a three-dimensional one. This change may affect distance metrics, and hence theoretical convergence; however, given a spatially discrete representation and a limited grayscale value range, it is possible to create a metric which will converge regardless of the graymap function (for instance, by weighting chromatic differences far less than spatial differences in the distance function).

Graymap functions are still generally required to convergent; otherwise, although the system as defined above (with the weighted distance function) is theoretically convergent, approximations with even a small error may still have very large grayscale differences when compared with the original. (This is unsurprising, as such large grayscale differences do not lead to large distances in the distance metric for the space. This distance function does not realistically model human perception of differences between images.)

2.2.2 Local Iterated Function Systems

A LIFS (Local Iterated Function System) is identical to an IFS except the domain of each transformation is not the entire space, but rather a subset of that space. The range of the transformation is therefore also a subset of the space. The transformations are not individually contractive with respect to the entire space; hence, the system may not be globally contractive and the results of the Contraction Mapping Theorem cannot be entirely relied upon. (As with an IFS, the result of applying the function system to a subset of the space is the union of the transformation results; hence, any parts of the space which are not covered by the range of any transformation will be absent from the result of a transformation.)

In [1], it is observed that a contractive LIFS may not have an attractor; further, it may have several attractors. If it has one or more attractors, it has one largest attractor, the superset of all other attractors

Attractors in an LIFS arise whenever an area in the range of a transformation affects the area of its domain, either directly (where the range and domain intersect) or indirectly through some series of transformations. In such spaces, the system forms a pure IFS and has a single attractor; hence, any seed which contains information which, through a series of transformations, affects the range, will give rise to that attractor.

It follows that, given a seed which is the union of all transformation domains, the largest attractor will be produced. (By extension, the universal set for the

space will also produce this largest attractor; in practice, it is often simpler to use the universal set than to find the union of the domains, especially if there are a large number of transformations.)

LIFSs used in image compression are frequently referred to as LIFSMs; the graymap function is treated separately from the topological transformations, analogously to IFSMs. Image compression algorithms are based on LIFSs, rather than IFSs, because doing so makes the burden of computation more reasonable; one need only find transformations whose fixed points approximate small pieces of the image, rather than the whole.

Maintaining the intensity axis as a single point, rather than a set of points, makes it impossible to guarantee that the largest attractor is produced. In practice, this does not appear to be a significant difficulty; the desired largest attractor is still usually found. An image compression system could perform a trial decompression to verify this, provided a constant seed image is always used at the start of iteration. If an incorrect attractor is found, some recovery action must be taken—either attempting compression again with slightly different parameters, or simply returning an error indicating the failure of the system to compress the desired image. As already mentioned, this is not a common problem with real-world images.

(It is an unproven conjecture of the author that, in the case of an image compression system, an attractor always will exist. An image compression system uses an LIFSM where the union of the ranges of the transformations covers the entire area of the image, and the domains of the transformations come from within the image. Under these conditions, it appears that it is impossible to generate a system which does not have an attractor.)

2.3 Other Theoretical Approaches

Fractal image compression may also be viewed as a special case of other compression technologies. Doing so gives some additional insight into how and why fractal compression works and also may give rise to hybrid approaches which have better performance. In [4], in addition to introducing convolution transform coding, a concise taxonomy of all of these theoretical approaches is presented.

2.3.1 Wavelet Compression

In [3], fractal image compression is investigated in terms of wavelets. It is shown that block-based fractal compression schemes are equivalent to self-quantizing a Haar subtree. Further, much of the effectiveness of fractal image coding systems is based on their ability to efficiently encode zerotrees.

By applying these observations to wavelets with basis functions other than the Haar basis function, Davis was able to improve their effectiveness. Further, Davis presented some statistical analysis which indicated that self-quantization should

be fairly effective at compressing textures in real-world images. (This analysis agrees with empirical observations of fractal image compression systems.)

2.3.2 Self Vector Quantization (VQ)

Vector quantization algorithms assume that image blocks may be adequately represented by one of a number of entries in a predetermined codebook, consisting of parameterized pattern blocks. This codebook is either agreed upon beforehand or must be transmitted as overhead data.

Fractal image compression may be viewed as VQ where the codebook is inherent in the image, consisting of the set of possible domain blocks. This codebook, rather than being transmitted separately or predetermined, is rebuilt by the decoder during the decoding process. Jacquin, in [6], recognized this when he introduced his algorithm.

2.3.3 Convolution Transform Coding

In [4], it is observed that fractal image compression may be regarded as a transform operator, based on the convolution of the range block with the set of domain blocks. This allows for a somewhat faster approach to compression (by performing fast convolution in the frequency domain) which is more suitable for software than for hardware. This also implies that fractal image encoders have, at their core, something very similar to a convolution engine; this is indeed true for the hardware design developed here, which could be transformed into a parallel convolution engine with only relatively simple changes.

Chapter 3

Algorithms for Fractal Image Compression

To compress an image with an LIFSM, it is necessary to find a set of contractive local transformations whose fixed points approximate the image. For compression to actually take place, this set must be expressible with less information than the image.

Decompression is performed by iterating the LIFSM over some seed image to produce an approximation of the attractor.

3.1 Characteristics of Fractal Image Compression Algorithms

Some characteristics are common to virtually all fractal image compression algorithms. Many of these are not found in most other image compression techniques; they are largely unique to fractal image compression.

Fractal image compression algorithms are inherently resolution-independent. Images may be decompressed at any desired target resolution, including those greater than the original resolution. The resulting images are smooth (not pixelated) and show "details" at their final resolution which were not part of the original. Obviously, any image compression technique cannot add information to the images; these details are figments of the compression and decompression techniques solely. Subjectively, they are often believable, especially at moderate levels of magnification.

As decompression at any desired resolution is possible, compression ratios must be computed at the original image resolution. Although this seems obvious, some proponents of fractal image compression wrongly compute compression ratios from higher-resolution decompressions and thereby create astoundingly good numbers. (Such results are sometimes referred to as "apparent compression ratios," rather than "compression ratios." They are still illusionary and unhelpful, except perhaps as marketing tools, as they may be crafted to be unrealistically high by performing decompression at large magnifications.)

Fractal image compression algorithms are extremely asymmetric with respect to processing time. Compression requires much more computation than decompression (often by orders of magnitude). This makes them more suitable for applications where compression is less frequent than decompression, such as shared image databases. The hardware proposed in this document is intended to help speed up compression; it is useless for decompression.

Fractal image compression algorithms are lossy, making them unsuitable for certain classes of work; in particular, many medical imaging applications require lossless compression.

Fractal image compression algorithms usually provide high compression ratios, often a bit better than JPEG compression at a similar perceived quality. (Image quality is generally subjective, imprecise, and often varies with the intended application of the images. In [5], for instance, it is shown that JPEG better preserves small, moderate-contrast features, especially at relatively high compression ratios.)

Fractal image compression algorithms are generally more suitable for use with "natural" images than with line art or similar classes of images. Further, the quality of compression varies significantly from image to image, and is not always discernible from obvious image qualities. (By contrast, JPEG compression displays artifacts around steep chromatic gradients in images; determining the relative content of high and low frequencies in an image suffices for determining its suitability for JPEG compression.)

3.2 A Generic Algorithm

Compressing an image using fractal image compression consists of finding an LIFSM whose attractor approximates the image in question. Since there are a theoretically infinite number of possible LIFSMs, only small classes of possible LIFSMs are considered; generally, those which are easiest to work with on a digital computer and those which require little information to describe (thus leading to high compression ratios).

This generic algorithm is typical of most approaches. A flowchart of the algorithm may be found in Figure 3.1.

The image is divided into a number of non-overlapping square range blocks of a fixed initial size. Domain blocks are constrained to have dimensions twice as large as those of the range blocks (that is, to each be a square with an area four times greater than that of a range block) and to fall within the image. The geometric transforms permitted are constrained to those affine transforms which map a given domain block to a given range block—four rotations and four reflected rotations. The intensity transformations are constrained to be linear functions.

For each range block, domain blocks are searched until one domain block (and orientation) may be nearly made to fit. The graymap function is determined using a least squares fit, and an error measure based on this least squares fit is

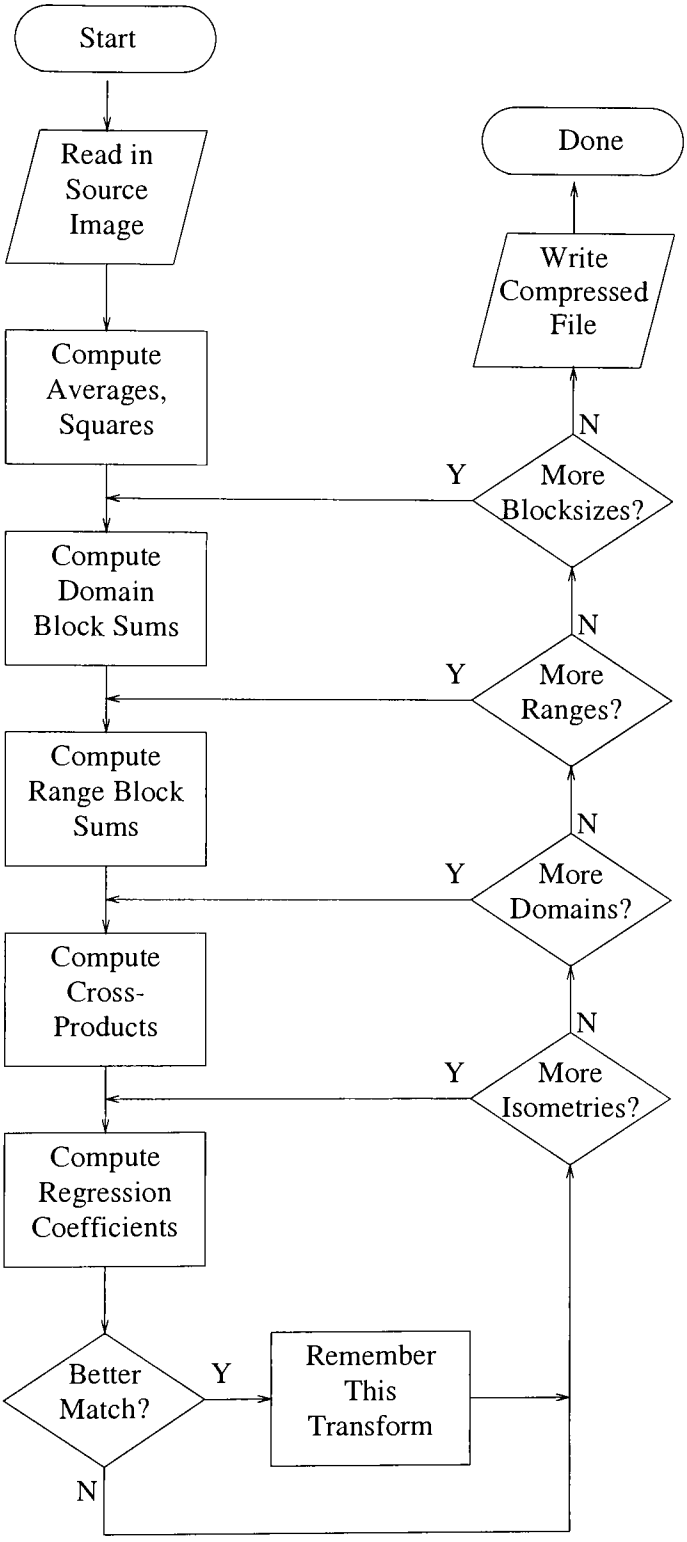


Figure 3.1: A Flowchart of the Generic Algorithm

computed. (The error measure used is generally the mean square error, or MSE; this implies that the distance metric used is not the Hausdorff metric as defined above; the theory is much simpler when the Hausdorff metric is used.) When a good fit (or the best possible fit) is found, it is recorded.

Usually, if no good fit is found for a block, it is subdivided and each smaller block is tested. (A quadtree approach is frequently used; this is referred to as quadtree decomposition, or QD.) Certain constraints also may be placed on the graymap functions; in particular, the allowed multiplicative constants are usually bounded to ensure the convergence of the decompression algorithm along the intensity axis.

The bit encodings of the transforms has a large impact on the compression ratios. These encodings vary considerably from system to system; therefore, they will not be covered in greater detail here. Applying some entropy encoding to the resulting transform encodings is common and gives a small increase in the compression ratio.

Appendix A gives example C programs which perform fractal image compression and decompression. The hardware is designed to essentially replace the core of this program (the do_block function); without too much difficulty, the program could be modified to drive the hardware.

Decompression is performed by applying the LIFSM defined in the compression to some seed image several times until it converges on (an approximation of) the attractor. Typically, this does not require many iterations, usually fewer than fifteen. The number of repetitions required appears to be largely independent of the target image.

3.3 Software Variations

3.3.1 Jacquin's Algorithm

Arnaud E. Jacquin was the first to propose a practical automatic (not requiring manual intervention or assistance) method of fractal image compression. His initial work was in a Ph.D. dissertation in 1989; a summary of that work is found in [6].

Jacquin's system used two fixed block sizes, selected based on an analysis of the range blocks. Range blocks were classified as either constant ("shade") blocks, modeled as a constant value; slowly-varying ("midrange") blocks, mapping to a transformation of a large domain block; or rapidly-varying ("edge") blocks, mapping to four transformations of four small domain blocks. The pool of domain blocks of each size was based on a sampling from the image, with blocks of constant brightness eliminated.

For midrange blocks, only one isometry was allowed and the graymap function was limited to a linear function with one of four given multipliers. For edge blocks, all eight isometries were allowed and six different multipliers were permitted for the graymap function. (It would seem that somewhat better results may have

been achieved if eight multipliers were permitted, rather than six, as doing so would not require any additional bits to encode.)

Jacquin used 8×8 range blocks for the large blocks and 4×4 blocks for the small ones.

Jacquin's algorithm, when compared with the generic algorithm presented previously, relies on categorizing the blocks to assist in finding good matches for them, rather than using a pure brute-force approach as above. Further, only two sizes of blocks are used, rather than a deeper quadtree as above. These differences are presumably to reduce the required compression time to a more reasonable amount, especially given the computer power available at the time of his work. (Moore's law suggests that current computers are approximately 100 times faster than they were at the time of the publication of Jacquin's paper.)

3.3.2 Classification of Blocks

Jacquin, and most other researchers, classify the set of range and domain blocks using some simple determination of their variance. Frequently, this is little more than separating out blocks which are essentially a solid gray (usually called shade blocks) value from those which contain noticeable gradients. Jacquin used a slightly more complex model, as described above. By classifying range and domain blocks, a directed search of the domain blocks may be done, requiring less time (by a constant factor).

Jumar and Jain proposed a much more complex system in [7], where 4×4 blocks were classified as either shade blocks or one of fifty-two varieties of edge blocks. Each of the edge blocks in a category were mapped to the same domain block, significantly cutting down on compression time by eliminating the search for domain blocks from the inner loop of the compression engine. They also employed a simple prediction algorithm for the shade and edge blocks which improved the compression ratios by omitting similar adjacent transforms. (Their approach is essentially a VQ scheme encoded as an LIFSM. A pure VQ approach might produce similar results with less overhead; they did not, however, investigate this possibility.)

3.3.3 Quadtree Recomposition

Quadtree recomposition (QR) builds a quadtree by starting with small blocks and then combining them as needed. This yields a significant speedup over a decomposition approach, as less computation is performed which is later rejected. QR is analyzed in some detail in [8] (and in [9], which has nearly identical material on the subject). In [8], QR reduced the compression times to about one-half to one-third of the times required for QD.

QR is a useful approach for software-based compression systems; however, it is less suitable for a parallel hardware implementation similar to that proposed here, as it initially requires a very large number of small blocks be computed. This would, in turn, require a very large number of repeated units in the pipeline. By

performing decomposition, the total number of units required at a given quadtree level is reduced. Further, unlike software, computations which are later ignored do not incur a time penalty; they may occur in parallel with those which are not ignored.

Since it was desired that the sample compression engine code presented here be similar to what would be used to drive the hardware, QD was used in preference to QR.

3.4 Genetic Algorithms

In [10], a genetic algorithm is used to direct the search for domain blocks. The authors found that doing so reduced the required number of comparisons substantially (by a factor of twenty in one case) while achieving results nearly as good as a complete search of the problem space.

3.5 Hardware and Parallel Approaches

Since hardware implementations may be (and generally are) inherently parallel, parallel software implementations can give useful insight into possible hardware designs.

3.5.1 Obvious Parallel Algorithms

Fractal image compression is easily parallelizable; the encoding of each range block is independent of the others. In [11], one example is given. This obvious general approach, treating the range blocks separately in parallel, is carried over in the design presented here.

3.5.2 Jackson's and Mahmoud's Parallel Approach

In [8], D. J. Jackson and W. Mahmoud describe an interesting parallel supercomputer implementation of Fractal Image Compression on a SIMD parallel machine, the nCube-2. Their approach is somewhat similar to the hardware approach described here, although it is implemented in software.

They used a single master control processor and a queue of slave processors which communicated to the master processor. The set of domain blocks was distributed among the slave processors. Range blocks circulate around a circular queue formed by the slave processors and are checked against the domain blocks. When a good match is found, the data is transmitted to the master and a new range block is inserted in its place.

Jackson and Mahmoud distributed the domain blocks, rather than the more obvious replication of them at all the slave nodes, due to memory constraints of the machine they were using. In the hardware proposed by this thesis, range blocks are distributed and domain blocks circulated through the set of range

blocks. This proves somewhat simpler to implement in hardware, as range blocks are both less numerous and smaller than domain blocks, allowing more parallelism.

3.5.3 Acken's, Irwin's, and Owens's ASIC Architecture

In [12], an ASIC architecture is described for fractal image compression. The architecture described is effectively a smart memory design; many small general-purpose processing elements are scattered throughout a memory array. (These processors are optimized for fractal image compression operations; however, it appears that they could do other operations with different microcoding.) The processors are arranged in a tree structure. Leaf nodes in this tree calculate sums and cross products for domain and range blocks; the non-leaf nodes use the results of lower nodes in the tree to determine a good mapping for the blocks. Through this tree structure, several levels of a quadtree decomposition of the image are computed simultaneously.

Several of the low-level design decisions they made are similar to those made here; in particular, the computations are performed using a bit-serial approach, similar to that used here. Similar constraints gave rise to these similar decisions; in both cases, the area consumed was a primary concern.

One important optimization used in their design is a reduction on the precision of some of the intermediate results in the calculations. They found that such reductions had very little effect on the quality of the output image, while reducing the time and area required for the computations. Such optimizations could probably be performed on the hardware described here with similar results.

Chapter 4

Hardware Design Overview

4.1 Goals and overview

The hardware design presented herein is aimed at substantially speeding up the encoding process by performing many of the repetitive tasks involved in finding transformations in parallel. It performs block matching using least squares linear regression on several blocks and orientations simultaneously. As it is a parameterized VHDL model, the number of range blocks processed simultaneously may be easily changed; this allows the design to be synthesized to suit the available area. (It would be desirable to have the maximum block size also configurable. Doing this would require a substantial amount of additional work and testing, as the widths of many of the signals depend upon this value. While design flexibility is an excellent goal, flexibility in this particular area required too much effort to be practical.)

The device is designed to be used in conjunction with a host processor which performs some of the higher-level computations and many bookkeeping tasks. Communication with the host processor is via several device control registers and a shared memory area to hold the source image.

4.2 General Notes and Some Conventions

A few important conventions are followed throughout this design. Signals, entities, and other VHDL objects are named in all caps, with underscores between words. In this document, references to specific VHDL objects are made with a typewriter font, `SAMPLE_VHDL_OBJECT`.

All signals are active high, with one important exception: `RESET`, which is for a power-on reset, is active low. This exception follows current practice and may lead to better synthesis for certain target architectures. `RESET` operates asynchronously. Not all sequential parts of the design are affected by `RESET`; some are reset at various times in the normal course of operation, and therefore do not require a power-on reset for proper operation.

4.3 Primary Components

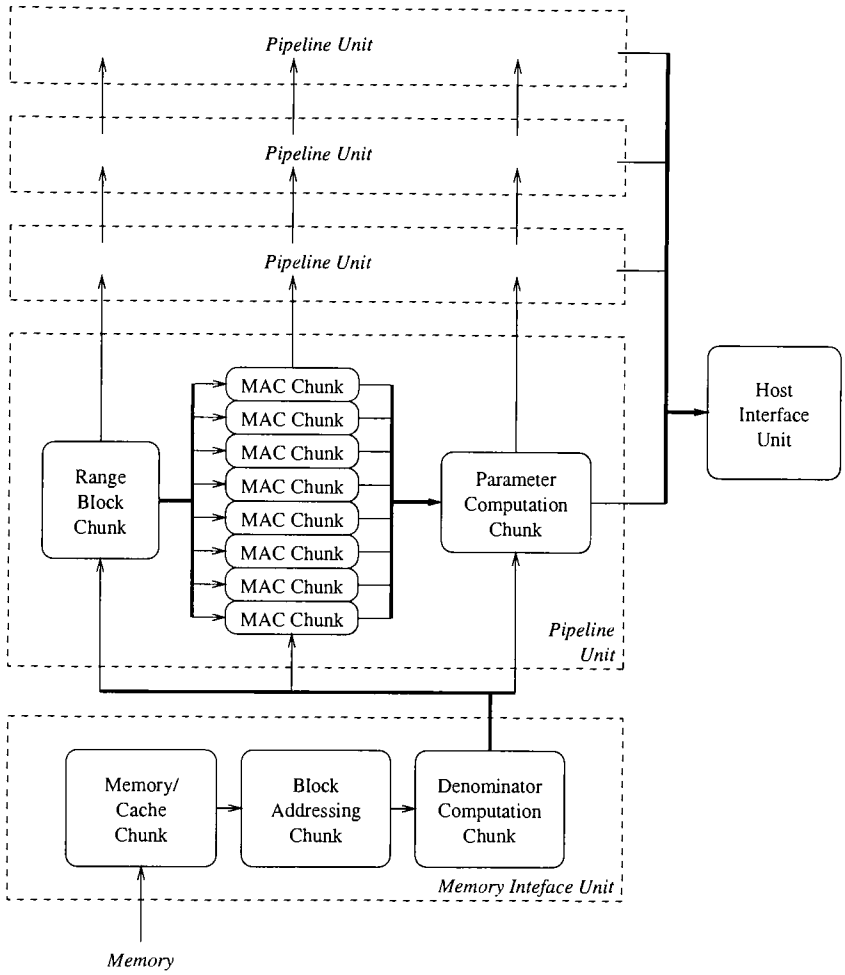


Figure 4.1: A Block Diagram of the Hardware

The hardware consists of three distinct kinds of blocks, called units: a single memory interface unit, a single host interface unit, and a number of pipeline units. Each pipeline unit performs comparisons of domain blocks with one range block; thus, ideally one should have as many pipeline units as there are range blocks of a particular size in an image.

The memory interface and pipeline units are further subdivided into smaller blocks, called chunks. Figure 4.1 shows an overview of the system architecture. (In this diagram, control signals have been omitted for clarity; the arrows indicate data flow only.)

In general, the design has been kept fairly simple (and thus small and relatively slow) to allow for a large number of units to be pipelined together. It is believed that performing many computations in parallel will be faster overall than performing fewer at a higher speed. Additionally, a fairly simple implementation

of the various parts yielded a design which is evenly matched in speed; no single component is a substantial performance bottleneck.

4.3.1 Main Memory Interface

The main memory interface is responsible for addressing memory shared between the host processor and the rest of the device. It is composed of three main components: a block addressing chunk, an interface and cache chunk, and a denominator computation chunk.

Block Addressing Chunk

The block addressing chunk generates the required addresses to access a single block within an image. This is controlled by three control signals: the starting address, the size of a block, and the length of a row in the source image.

Additionally, the block addressing chunk performs pixel averaging for domain blocks. The block addressing chunk is implemented, in VHDL, as a fairly straightforward state machine with some auxiliary registers for performing addressing. More details of the design are found in Section 5.2

Memory Interface and Cache Chunk

This chunk performs the actual interfacing with the memory bus. Currently, it is assumed that the memory is sixteen bits wide; this block performs the byte extraction as necessary. It also maintains any memory cache used; in the current design, this is a very small four byte cache, which is only sufficient to prevent repeated accesses to the same memory word, such as would otherwise occur when averaging four adjacent pixels during the reading of domain blocks.

If the memory interface were the limiting factor on system performance, a larger (but still relatively modest) cache capable of holding any single domain block would significantly reduce the memory bandwidth requirements. A further reduction could be realized by having an entire row of domain blocks in cache at once; in this last case, an average of about one byte per domain block would be fetched from memory during operation.

Separating these parts from the rest of the system allows the design to be modified for use with various system architectures easily. The specifics of the cache is very heavily related to the precise bus design and the hardware speed, and is therefore likely to change when the memory interface changes. Keeping these two in the same chunk minimizes the difficulty of adapting the design for other environments.

Details on the design used are found in Section 5.1

Denominator Computation Chunk

This chunk performs computations which are used by all the parameter computation units in the pipeline. This chunk is also responsible for high-level addressing

of domain blocks—that is, for automatically iterating over all the blocks in the image.

The two functions of this chunk are largely separate, and could easily be implemented as two chunks. As neither is very complex, however, it was simplest to combine them and avoid an additional level of interface.

Section 5.3 describes this chunk in greater detail.

4.3.2 Pipeline Unit

The core of the system is a series of pipeline units; each unit compares a single range block with a stream of domain blocks. The units each consist of several chunks: a range block chunk, a parameter computation chunk, and eight multiply-accumulate (MAC) chunks. Ideally, enough pipeline units should be present in an implementation to be able to hold the maximum number of range blocks of some size in an image to be compressed.

Range Block Chunk

The range block chunk stores one range block from the image and provides the MAC chunks with pixels from the block in various orderings (which correspond to the eight possible isometries for mapping one block to another). It consists of a small memory array and the required addressing hardware. The memory array is actually implemented in a separate file, to allow one to easily use a pre-defined memory component for an actual implementation technology. (Synthesis tools generally produce inefficient circuitry for relatively large blocks of memory.) Two versions of this file were developed: one as a placeholder for synthesis, containing instantiations of architecture-independent register arrays; and one as a placeholder for simulation, containing an algorithmic description of the memory. (This was done because the synthesizable version was significantly slower for simulation.)

Key to the range block chunk, but also important elsewhere, is the maximum block size; larger blocks require larger memory arrays. A block size of 32×32 is currently used and appears to be a reasonable value. (Each range block chunk contains enough memory to store one block; for a 32×32 block size, this amount is 1,024 bytes.)

Details on the range block chunk are found in Section 6.2

Multiply-Accumulate Chunks

These chunks combine an eight bit multiplier and a twenty-six bit accumulator. Multiplication is unsigned and is performed using the usual shift and accumulate approach (taking eight clock cycles per multiplication). Faster multiplier designs are not useful here, as the range block units would have to have more than one read port and the main memory interface would need to supply data faster, both of which add substantially to the complexity of the system. Further, it is likely

that the bandwidth to the main memory would not allow a substantial increase in operational speed without a lot more caching of memory data.

The MAC chunks are equipped with a double-buffered output; this allows the parameter computation chunk to operate on one set of numbers while the next set is being computed in the MAC chunks.

Since all the MAC units are implemented in a single pipeline, one of the operand shift registers (that for the domain block data) is actually implemented as single bit slices in each MAC unit.

Parameter Computation Chunk

The parameter computation chunk is responsible for computing a distance measure for the mapping. This error measure is based on the mean square error when linear regression by least squares is applied, given certain constraints (primarily requiring that the resulting transformation be contractive in all dimensions).

The mathematical background and design of this chunk is more complicated than that of the other chunks; the reader is referred to Section 6.1, where it is explained in detail.

4.3.3 Host Interface Unit

The host interface provides for communication between the host processor and the hardware. It also generates certain control signals which are necessary for the operation of the system. The host interface consists of a number of sixteen-bit registers; currently, no interrupt capability exists, although that could be added without great difficulty. (This was not included because the specifics of the interrupt system are highly dependent on the specific host processor used and because it was not helpful in the simulations of the system.)

Specifics for the host interface are found in Chapter 7.

4.4 Testing and Support Code

A few VHDL entities were developed for testing the system. These are not intended to be synthesized—they exist only to aid in simulating the system operation.

4.4.1 Clock and Reset Generator

A very simple clock and reset generator was implemented. The simulated clock operates at 10 MHz. This speed was chosen because it made mental conversions of time and clock tick counts simple; an actual implementation of the design using current technology would probably run at a higher clock speed.

4.4.2 Simulated Memory

A simulation of the shared memory was developed; this simulated memory is automatically loaded from a file when the simulation is started. The bmp2memfile program, described in section D.1, creates this file.

Chapter 5

The Memory Interface Unit

The memory interface unit is divided into three components: the memory cache chunk, the block addressing chunk, and the denominator computation chunk. The memory cache chunk is responsible for talking to the main memory system and for any data caching. The block addressing chunk generates the appropriate addresses for accessing blocks of an image and performs some important computations on domain blocks (pixel averaging and the computation of the quantities needed for parameter computation). Finally, the denominator computation unit forms the interface between the memory addressing hardware and the pipeline units by automatically adjusting the block addresses as needed and by computing the denominator value required by the parameter computation unit ($n\sum y^2 - (\sum y)^2$). It also computes the parameters associated with range blocks when they are loaded ($\sum x$ and $(\sum x)^2$).

This separation has several important advantages. It makes the design more manageable and understandable; a single unified unit would be significantly more complex than any chunk. It makes improving the cache system much easier, as the cache logic is separate from the other parts. It also makes the design much easier to modify for different memory bus designs.

Separating the block addressing and the denominator computation allows changes to be made more easily to the pipeline (and, in particular, to the parameter computation strategy) without changing the logic to address blocks.

The cache and the memory interface were not separated because they are inherently closely related. The amount and organization of the cache is largely determined by the specifics of the memory interface.

5.1 Memory Cache Chunk

5.1.1 The Sample Memory Interface

The memory interface currently used in this design is a simple, representative sixteen-bit wide bus. While typical of many standard bus designs, and especially similar to an MC68000 bus, it does not attempt to adhere to any specific protocol. A modern implementation would probably need a more complex external bus,

and thus substantial changes to the implementation. The general approach to the design described here ought to be adaptable to a wide variety of situations, however.

Arbitration signals for the memory are BUS_REQUEST and BUS_AVAILABLE. When the system needs some datum from memory, it first asserts BUS_REQUEST; some external logic should assert BUS_AVAILABLE when the device has the bus. The device keeps BUS_REQUEST active throughout its memory operation (which is always a single word).

The memory is addressed using ADDRESS_OUT, strobed by ADDRESS_STROBE. The memory is byte addressed with a 32-bit wide address bus; however, since only word (two byte) transfers are performed, the least significant bit of the address is omitted from the physical memory bus.

The external memory asserts MEM_READY after it puts the requested data on DATA_IN. Once the device has strobed in this value, the memory cycle terminates and control of the bus is relinquished.

All inputs are sampled on the rising edge of the clock, and outputs change on the rising edge of the clock; hence, this interface may be characterized as semi-synchronous. No precise timing requirements are presented for the sample memory interface bus; it is intended primarily as a placeholder to enable testing.

Since the device never writes to memory, no protocol for writes is defined in this sample interface.

5.1.2 System Interface Logic

The memory cache chunk must provide individually addressed bytes to the rest of the system. The interface used is synchronous.

The system provides a thirty-two bit address on ADDRESS_IN, and a flag indicating a valid request (DATA_OUT_NEEDED). The cache unit should respond with the requested data in DATA_OUT and set the DATA_OUT_HERE flag.

In the sample design, the logic to drive this interface is entirely combinational. It simply determines if the data is in the cache (based on the cache tags), and extracts it if it is present. If the data is not in the cache, a flag indicating this is asserted, which starts the main memory interface state machine. Eventually, when a memory cycle completes, the data will be placed in the cache and simultaneously propagated to the rest of the system.

5.1.3 Memory Interface Logic

A state diagram of the memory interface logic is shown in Figure 5.1. The BUS_REQUEST state is active whenever the memory bus is wanted but not currently granted; the MEMORY_WAIT state is active while the system waits for the memory to respond.

When a read occurs, the data goes into one of two word-sized caches blocks. A pure FIFO algorithm is used; due to the nature of the memory accesses, this is ac-

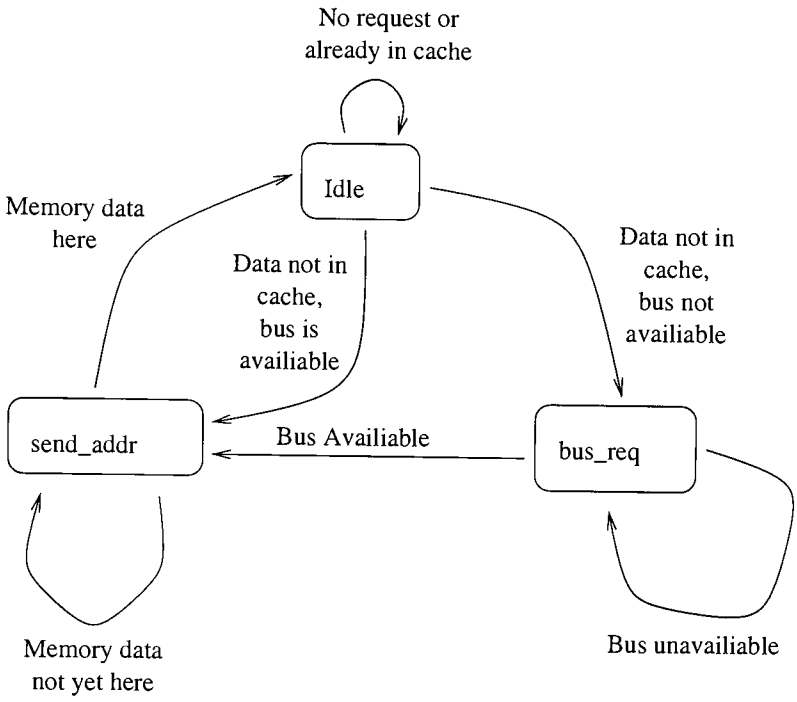


Figure 5.1: Memory Interface State Diagram

tually an optimal strategy when reading domain blocks for this (small) cache size. A further discussion of the cache in this design may be found in Section 10.1.1.

In addition to updating the caches, the result of a read is propagated to the block addressing chunk immediately. (The alternative, only passing on values from the cache registers, adds an additional wait state to memory accesses and thereby slows down the system substantially.)

5.2 Block Addressing Chunk

The block addressing chunk is primarily responsible for generating memory addresses to access a block within an image. The image is assumed to be uncompressed, so the addressing is fairly straightforward. Specifically, the image is stored in a rectangular array in memory, perhaps with padding at the ends of the rows of the image. (Throughout this document, it is assumed that the image is in row-major ordering with the origin at the upper-left corner of the image. This is primarily a conceptual convention; column-major ordering poses no operational difficulties to the hardware, but does change the interpretation of a few inputs and outputs.)

In addition, the block addressing chunk performs averaging on four adjacent pixels when reading domain blocks. When loading range blocks, no averaging occurs.

Figure 5.2 shows the state diagram for the block addressing chunk. The

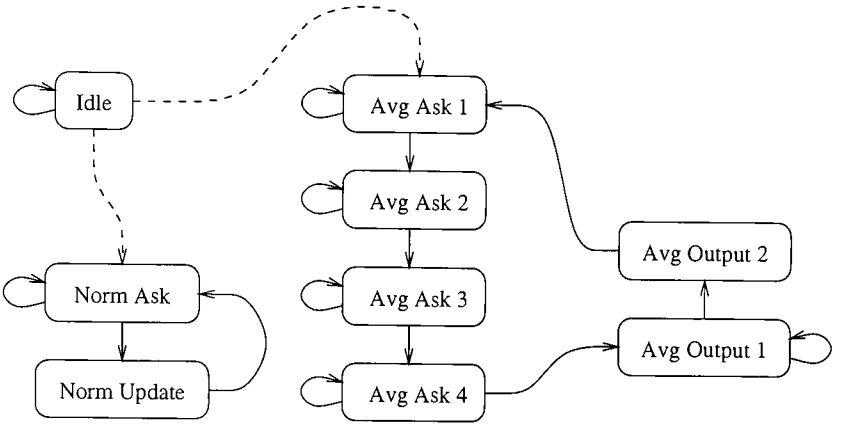


Figure 5.2: State Diagram for the Block Addressing Chunk

dotted arrows, from the IDLE state to the NORM_ASK and AVG_ASK1 states indicate a typical path; in the actual implementation, the AVG_ASK1 and NORM_ASK states may be entered at any clock cycle when the START signal is active.

The NORM_ASK and NORM_UPDATE states are used when loading a range block; the others are used when comparing blocks (e.g. when reading domain blocks and averaging pixel values). During the various _ASK states, the system loops until the memory is ready; during AVG_OUTPUT1, the system loops until the denominator computation chunk is ready for another value.

5.3 Denominator Computation Chunk

The denominator computation chunk fulfills two main purposes: it provides high-level addressing of blocks (as opposed to addressing pixels within a block, which the block addressing chunk does), and it computes a few values which are used for parameter computation. These two activities do not interact much; they could fairly easily be implemented in separate chunks. As they are both fairly simple and straightforward, this was not done; the complexity of a single chunk containing both is not too great and avoids an additional interface which must be maintained.

5.3.1 High-level Block Addressing

The state machine which performs the high-level block addressing is shown in Figure 5.3. There are two primary paths in this system—one, through LOAD_START, for loading range blocks, and one for checking denominators. The M1 through M8 states are to ensure the various multipliers, in the multiply-accumulate units and in the denominator computation itself, have enough time to operate. Although the arrows are not explicitly shown, the system goes invariably from state M1 to M2 and so on to M8.

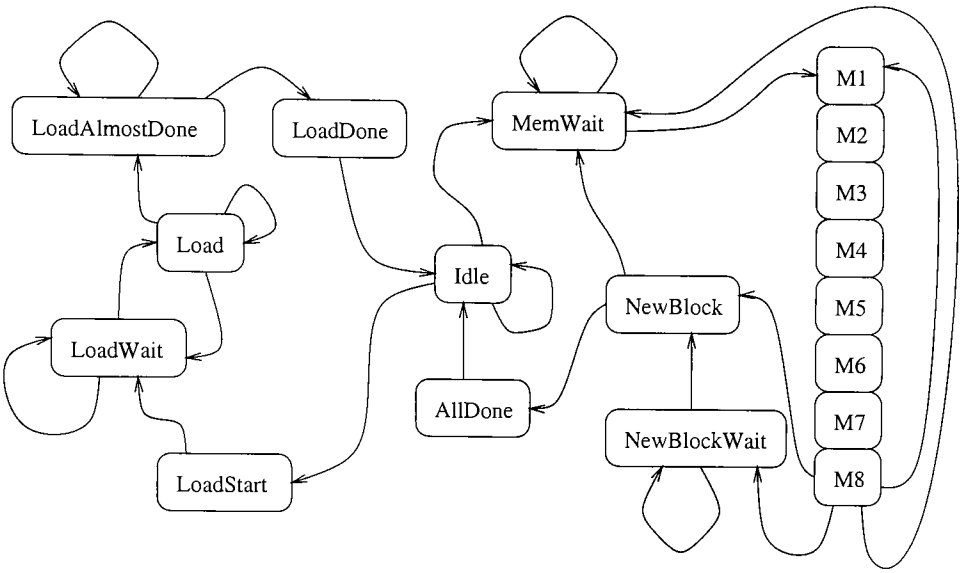


Figure 5.3: High-Level Block Addressing State Diagram

A series of addressing counters, updated in the `NEW_BLOCK` state, control block addressing. A single counter is used to count the pixels within a single block.

5.3.2 Denominator Computation

The denominator computed is $n\sum y^2 - (\sum y)^2$. The computation is performed using a combinational subtracter and shift-and-add multipliers; it effectively adds ten cycles to the parameter computation time required. (The entire process requires eighteen cycles, but eight of those are overlapped with the eight cycles required for the multiply-accumulate units to perform a multiplication.)

Also computed as part of this are $\sum y$ and $\sum y^2$, which are also required by the parameter computation. Through double-buffering, the parameter computations and multiply-accumulate steps overlap. For large blocks, larger than about 10×10, the limiting factor is the multiply-accumulate time; for smaller blocks, it is the parameter computation time.

This hardware is also used to compute $(\sum x)^2$ when range blocks are loaded.

A dataflow diagram of the denominator computation is shown in Figure 5.4.

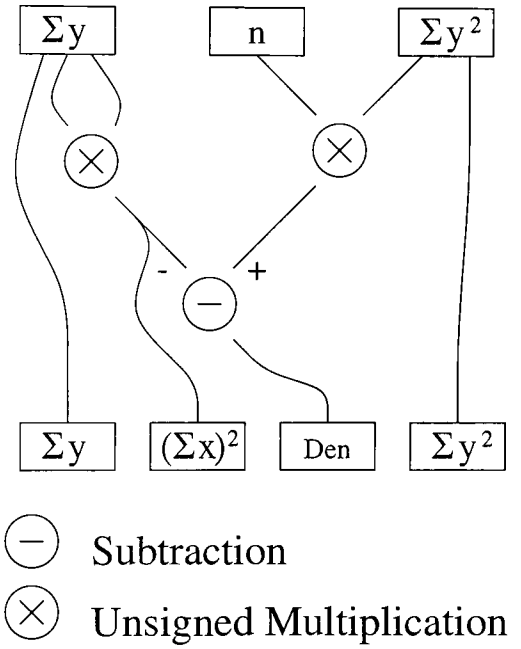


Figure 5.4: Dataflow for the Denominator Computation

Chapter 6

The Pipeline Unit

The pipeline unit is composed of a range block chunk, eight MAC chunks, and a parameter computation chunk. The parameter computation chunk is, by far, the most complex of these.

6.1 The Parameter Computation Chunk

6.1.1 Mathematical Basis for Parameter Computation

In least squares linear regression, given a set of range points $x_1, x_2, \ldots, x_n$ and a set of domain points $y_1, y_2, \ldots y_n$ (which, in this case, are averaged values), an approximation is given by $x_i = ay_i + b$, where

$$a = \frac{n\sum xy - \sum x \sum y}{n\sum y^2 - (\sum y)^2}$$

and

$$b = \frac{\sum x - a\sum y}{n}$$

The mean square error (MSE) is given by

$$MSE = \frac{\sum[x - (ay + b)]^2}{n}$$

Expanding this expression, one finds that

$$\begin{aligned} MSE &= \frac{\sum[x^2 - 2x(ay + b) + (ay + b)^2]}{n} \\ &= \frac{\sum(x^2 - 2axy - 2bx + a^2y^2 + 2aby + b^2)}{n} \\ &= \frac{\sum x^2 - 2a\sum xy - 2b\sum x + a^2\sum y^2 + 2ab\sum y + nb^2}{n} \\ &= \frac{\sum x^2 + a(2b\sum y - 2\sum xy + a\sum y^2) + b(nb - 2\sum x)}{n} \end{aligned}$$

If the expression for b is substituted in, the result may be reduced to

$$MSE = \frac{n\sum x^2 - (\sum x)^2}{n^2} - 2a\left[\frac{n\sum xy - \sum x\sum y}{n^2}\right] + a^2\left[\frac{n\sum y^2 - (\sum y)^2}{n^2}\right]$$

Finally, when the expression for a is substituted into this equation, it may be reduced to

$$MSE = \frac{1}{n}\left[\sum x^2 + \frac{2\sum xy\sum x\sum y - (\sum x)^2\sum y^2 - n(\sum xy)^2}{n\sum y^2 - (\sum y)^2}\right]$$

The parameter computation chunk computes an expression derived from this for every isometry and keeps the one with the lowest error, provided the a parameter satisfies $-1 \leq a \leq 1$. It would be desirable to also determine the MSE with $|a| = 1$ when the best a satisfies $|a| > 1$ and store the pairing if, with $|a| = 1$, the MSE is still better than the previous best. This is not actually done, as doing so would require a great deal of additional hardware, and the improvement in compression is usually insignificant.

The quantity $D = n\sum y^2 - (\sum y)^2$ is computed by the memory interface unit, and is passed through the pipeline of the pipeline units. If this quantity is zero, then the domain block is (when averaged) a solid, constant value; in this case, it should be discarded from the search. (It is assumed that range blocks which are adequately modeled as shade blocks are handled by the host processor separately; therefore, comparing a non-shade block to a pure shade block is useless. Not discarding such a domain block may result in an attempt to divide by zero if it is determined to be a new best block, hence the requirement to discard. The actual hardware uses a slightly different approach to the problem, and avoids the division altogether; because of this, it is not necessary for the hardware to discard such domain blocks.)

The parameter computation unit must (for each isometry) determine if $|a| \leq 1$; this is equivalent to testing whether $|n\sum xy - \sum x\sum y| \leq n\sum y^2 - (\sum y)^2$, which is actually done. If the pairing passes the unit then determines if

$$D_s\left[2\sum xy\sum x\sum y - \left(\sum x\right)^2\sum y^2 - n\left(\sum xy\right)^2\right] < rD$$

If so, it stores new values for $r = 2\sum xy\sum x\sum y - (\sum x)^2\sum y^2 - n(\sum xy)^2$ and $D_s = D$. (Here, r/D_s is a measure of the relative MSE difference). Note that r may become negative; thus, the comparison and multiplications must be signed. $n\sum y^2 - (\sum y)^2$ can never be negative, however.

6.1.2 Implementation

A block diagram of this operation is shown in Figure 6.1. The implementation used is a pipelined serial design, where one bit proceeds down the pipeline at each clock cycle. Pipeline registers are located only at the multipliers; the adders and comparators are essentially combinational and do not contribute to any latency.

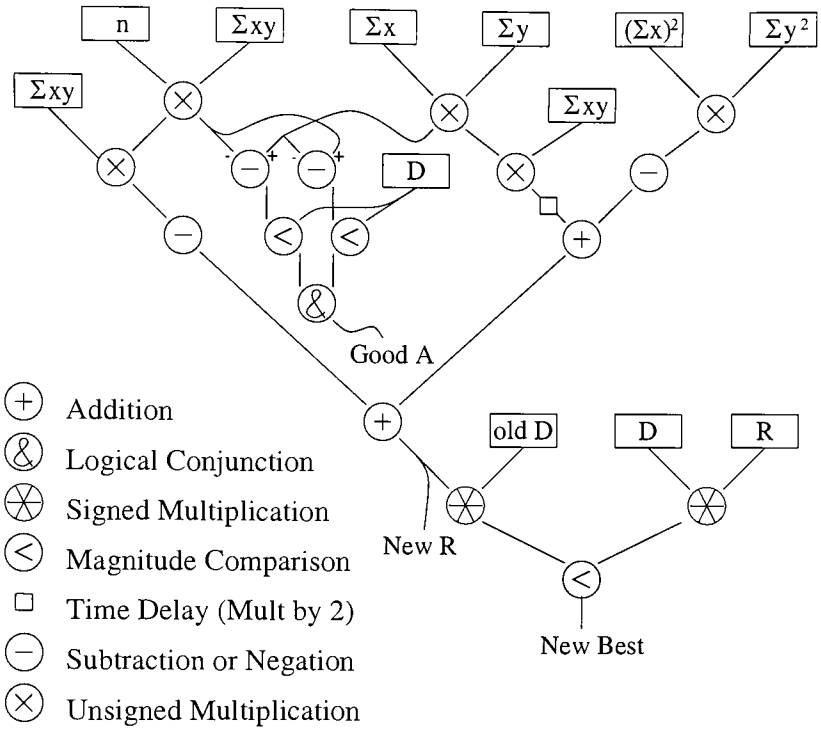


Figure 6.1: A Block Diagram of the Parameter Computation

(These components are not completely combinational, in that there is some state associated with them—for instance, carry bits in the adders. They do not operate as separate pipeline stages, however.) This approach was used because it requires relatively little area while giving adequate performance. (With the current design, the parameter computation becomes a performance bottleneck when the range block size is less than about 10×10; for larger blocks, the time required to perform the comparison with the domain block dominates. These two operations take place in parallel. The exact transition point depends on the maximum block size allowed by the design, which in turn determines the width of the computations in the unit. Currently, 32×32 is the maximum value.)

Multiplications throughout the parameter computation unit are performed by a shift-add approach. Where signed multiplication is required (the two final multiplications), a radix 2 Booth multiplier is used. In the situations where an unsigned multiply is followed immediately by a negation, these two operations are combined by using a shift-subtract design rather than a shift-add design; this is not reflected in the diagram in any special way.

If a parameter comparison is good—that is, if the new mapping is better than the previous best—the system must store four pieces of information: the new D_s, the new r, the coordinates of the domain block (stored as the starting address of the block), and the particular isometry. Only the last two of these need be visible to the host processor; the values of r and D_s are of little concern to the driver program. Since an actual MSE value and a and b regression parameters

are not computed, the host processor will need to determine these for each block. This is not too great of a computational burden, as it only must be done for one mapping per range block. The alternative, computing them in special-purpose hardware, would require significantly more circuitry than the present design, and would end up reducing the performance of the system by allowing fewer pipelined comparison units on a chip. (In particular, computing the actual values for these would require circuitry for division, which does not fit well with the bit-serial computations performed in this design.)

The parameter computation chunk also buffers the required values from the memory interface unit and passes them on to the next pipeline stage. Further, it produces values for the local host registers, which are multiplexed to the host when requested.

6.2 The Range Block Chunk

The range block unit is conceptually straightforward. It consists of a small memory block, sufficient to hold one range block, and appropriate addressing circuitry. The addressing circuitry required to produce the eight possible isometries is not too complex; various permutations of two counters and two subtracters are all that is required. Each pipeline unit maintains a separate set of these counters; it is thought that the overhead of passing the values along the pipeline would outweigh the overhead of the counters themselves. The current approach is also somewhat simpler conceptually.

The actual implementation is a little less clear, due to constraints in synthesis. Synopsys, like most current synthesis tools, is not especially adept at synthesizing anything other than small blocks of memory. For this reason, a separate file contains an entity for the actual memory block. It is anticipated that an actual device would use a pre-designed memory component; having the description in a separate file allows this to be done easily. As a generic placeholder, an array of DesignWare memory components is instantiated in this file to allow synthesis. The resulting synthesized hardware is not very area efficient, however; a flip-flop is used for each bit in the memory, rather than a normal static or dynamic RAM cell.

Table 7.1.2 lists the eight isometries and the addressing which is used to generate them.

6.3 The Multiply-Accumulate Chunks

These are quite straightforward; the overview in Section 4.3.2 should be sufficient for the reader to understand them.

Chapter 7

The Host Interface Unit

7.1 Description

The host interface allows the host processor to control the device and to read data back after the computation is done. It consists of several memory-mapped registers, some of which are read-only and some of which are also writable. The registers are divided into two sets: global registers, which apply to the system as a whole, and local registers, which apply only to one particular pipeline unit. There are sixteen address locations for each set of registers, but not all are actually implemented.

All registers are nominally sixteen bits wide, although not all bits may be operable; bits which are not used are always read as zeros and are ignored on writes. In general, if a register is partially implemented, the least significant bits are the ones implemented.

The registers are outlined in Table 7.1; a description of the individual registers follows.

7.1.1 Global Registers

Except for certain bits in the Mode/Control register, all global registers are writable. For correct operation, there are some restrictions on when they may be written. These restrictions are not enforced by the hardware and must be observed by the host software.

Mode/Control Register

This register determines the global mode of the system (whether loading range blocks or computing domains) and commences any operations. It also contains flags indicating the system status. Only two bits are used.

The load bit, bit 1, determines the mode the system is in; writing a value of 1 to this bit indicates that range blocks are to be loaded, while a zero indicates that comparisons are to be performed. This bit should not be changed during the entire loading process (e.g. it should remain at 1 throughout).

$A_{3..0}$	Global Reg ($A_4 = 0$)	Local Reg ($A_4 = 1$)
0	Mode/Control	Flags
1	Block Size	Best Isometry
2	Block Pixels	Best Address (MSB)
3		Best Address (LSB)
4		Best r (MSB)
5		Best r
6	Image Row Length	Best r
7	Image Width	Best r (LSB)
8	Image Height	Best D_s (MSB)
9	Pipeline Unit Select	Best D_s (LSB)
A	Base Address (MSB)	$\sum x$ (MSB)
B	Base Address (LSB)	$\sum x$ (LSB)
C		
D		$(\sum x)^2$ (MSB)
E		$(\sum x)^2$
F		$(\sum x)^2$ (LSB)

Table 7.1: The Host Registers

The start bit, bit 0, commences a range block load or a series of comparisons when a 1 is written to it. The implementation allows the writing of both the mode bit and the start bit at the same time; if the mode is changed, starting will commence the operation in the new mode. Technically, the start bit is a write-only bit; reading the bit produces the busy bit. Since the two are closely related, no real confusion should result. Setting the start bit while an operation is in progress is not allowed. (It probably would cause the operation to start again from scratch, but this is untested and possibly untrue in some cases.)

The busy bit, also bit 0, indicates that the system is working, either loading or comparing as the case may be. This is a read-only bit. This bit is cleared when the first parameter computation chunk has finished; the others may still be busy for a time afterwards. (This allows the host processor to read early results while the later ones are still being computed.)

There may be a delay of a few clock cycles between writing a one to the start bit and the busy bit being set. It is therefore recommended that a small delay be inserted between a write and a subsequent read of the register.

Additional bits could be implemented in the future if some application required them. Some possible examples include a last pipeline unit busy bit, a reset bit, and a halt bit.

Block Size Register

This six-bit register must be loaded with the size of the desired block, less one; thus, for 8×8 blocks, a value of 7 must be loaded. The largest supported block size is (currently) 32×32. This should not be changed while any operation is in

progress; further, it should have the same value when comparing blocks as it had when loading the range blocks for the results to be valid.

Block Pixels Register

This eleven-bit register should be loaded with the number of pixels in a block—for example, for a 8×8 block, it should be loaded with 256. This register should be changed under the same circumstances as the Block Size Global Register.

Image Row Length Register

This sixteen-bit register specifies the length, in memory, of a row in the image; this includes any padding which may be present in memory but not part of the image proper. It is used for address computations. (It is assumed that the image is stored in an uncompressed row-major array of bytes.)

This register generally should be set only once per image and not changed subsequently. It must not be changed while any operation is in progress.

Image Width Register

This sixteen-bit register counts the number of (overlapping) blocks present in a row of the image.

This register should not be changed while block comparisons are happening.

Image Height Register

This sixteen-bit register counts the number of (overlapping) blocks present vertically in the image.

This register should not be changed while comparing blocks.

Pipeline Unit Select Register

This sixteen-bit register chooses which pipeline unit will be loaded and which one will provide values for the local registers. Pipeline units are numbered starting with zero; the results of loading this register with a number greater than the greatest pipeline unit is undefined (and thus doing so should be avoided.)

This register must not be changed while a range block is being loaded; it may be changed at any time during block comparisons, however.

Base Address Register

This thirty-two bit register specifies the base address of either the image (for comparing blocks) or the specific range block (for loading a range block). It may be changed at any time without ill effect.

Other Possible Registers

Other registers may be helpful in some applications; for instance, a read-only register which contains the number of pipeline units would allow driver software to automatically adjust to various implementations of the design.

7.1.2 Local Registers

All local registers are read-only. Many may not be of much use during general operation of the device, but are provided for the software in case they actually turn out to be useful. These unnecessary registers may be removed from the system, or replaced with more useful ones. (They are not particularly helpful for simulations, as the simulator allows one to easily examine the signals directly.)

Flags Register

This register contains flags which are local to the pipeline unit. At present, there is only one such flag, the idle flag, which is set whenever the parameter computation chunk of the pipeline unit is not operating. This allows the driver software to ensure that the pipeline unit has finished the last comparison before reading the results.

Best Isometry Register

This three-bit register contains a code for the isometry of the best match found in the search. The translation of the isometries is provided by Table 7.1.2; all rotations are clockwise, and all reflections are about the horizontal axis. (This table assumes a row-major ordering of the pixels, from the upper-left corner to the lower-right corner of the image. Other arrangements would lead to different geometric descriptions.) This register is automatically cleared to zero when a new range block is loaded.

Code	Geometric Definition	Algebraic Description
000	No Isometric Transformation	$D_{x,y} \rightarrow R_{x,y}$
001	Reflection, Rotation by 90°	$D_{x,y} \rightarrow R_{y,x}$
010	Reflection, Rotation by 180°	$D_{x,y} \rightarrow R_{n-x,y}$
011	Rotation by 270°	$D_{x,y} \rightarrow R_{n-y,x}$
100	Reflection	$D_{x,y} \rightarrow R_{x,n-y}$
101	Rotation by 90°	$D_{x,y} \rightarrow R_{y,n-x}$
110	Rotation by 180°	$D_{x,y} \rightarrow R_{n-x,n-y}$
111	Reflection, Rotation by 270°	$D_{x,y} \rightarrow R_{n-y,n-x}$

Table 7.2: The Isometry Codes

Best Address Register

This register contains the address of the first pixel in the best domain block for the range block; it is a fairly simple affair to convert this into a set of coordinates. This register is cleared to all zeros when a new range block is loaded; hence, it is recommended that the image data exist elsewhere in the address space.

Best r Register

This register gives the r parameter (as computed by the parameter computation unit) for the best domain block. This value is most useful for debugging and testing the chip design, rather than being useful for any host-based computation, and may be removed from the design eventually.

Best D_s Register

This register gives the denominator used in the parameter computation unit. Like the best r register, it is of little use to the host system under normal operation.

$\sum x$ Register

This register contains the sum of the pixels contained in the range block. This is computed when the block is loaded, and is useful to the host for determining regression parameters or other simple image processing.

$(\sum x)^2$ Register

This register contains the square of the sum of the range pixels, computed when the range block is loaded. This value is also useful for finding regression parameters.

7.2 Implementation

The implementation of the host interface is fairly straightforward; unlike the other units, it is not a composition of several chunks. The global registers form a specialized register file, the outputs of which are always available to the system. The local registers are multiplexed from the various pipeline units, as selected by the pipeline unit select global register. (The actual selection of which specific local register to read is performed by each pipeline piece. This simplifies the wiring overhead of the system somewhat.)

If the VHDL synthesis tool supports it, the local register selection could be simplified by the use of internal tri-state signals. Not all target architectures support such signals, however, and not all VHDL synthesis tools are capable of using tri-state signals. For these reasons, the current design does not use tri-state signals.

Chapter 8

Testing and Verification

Testing was performed in several stages. Algorithmic testing was performed using the compression and decompression program written in C and contained in Appendix A. Verification of the basic operation of the chunks was performed individually; care was taken to ensure that all major points of operation were tested, although no attempt was made to formally cover every possible condition. Verification of the design as a whole was performed by simulating the compression of a (quite small) image and comparing the result to that generated by the C program.

This hybrid testing approach was used for two main reasons. First, it allowed the algorithm to be perfected without undue difficulty in constantly changing the hardware design. Second, simulating the hardware takes a very long time and is therefore impractical for all but the smallest images. (For the 80×56 test image used, each hardware simulation required about eighteen hours.)

8.1 Algorithmic Testing

The program presented in the appendix is the result of several iterations of development, experimenting with various distance measurements and other algorithmic strategies. In its current state, it is typical of many fractal image compression systems in the literature. A few simplifications are made, particularly in the method of handling images whose dimensions are not an even multiple of the block size used. (Currently, such images are cropped; this is hardly acceptable for general-purpose use.).

A summary of some results of using this program with various images is presented in Table 8.1. A summary of results for a single image with various parameter values is in Table 8.1. In these tables, two compression ratios are given; the ratio in parentheses is the compression ratio after the encoded image has been compressed using gzip. (All the images are 256×256 grayscale images; all compression ratios are computed based on an input Windows .BMP file, which is 67382 bytes long—including 1846 bytes of header information.)

Note that the compression ratios achieved are not the best possible with frac-

tal image compression, as the transformation parameters are encoded with greater precision than is necessary. It is expected that, without significant changes in image quality, the compression ratios (prior to entropy encoding with gzip) could be at least doubled. (The entropy encoding step, in this case, would probably not be quite as effective, although it would still improve the compression.)

Image	Options	Compression	Output
sunset (Figure 8.1)	-a 1 -b 32 -c 25 -m 4	3.26:1 (6.17:1)	Figure 8.5
textl (Figure 8.2)	-a 1 -b 32 -c 25 -m 4	1.38:1 (4.42:1)	Figure 8.6
chapel (Figure 8.3)	-a 1 -b 32 -c 25 -m 4	1.54:1 (2.59:1)	Figure 8.7
coke (Figure 8.4)	-a 1 -b 32 -c 25 -m 4	2.82:1 (5.21:1)	Figure 8.8

Table 8.1: Summary of Test Results with Various Images

Max a	Blocksize Range	MSE Cutoff	Compression	Output
1.0	32 – 4	25	3.26:1 (6.17:1)	Figure 8.5
1.0	32 – 4	10	2.07:1 (3.53:1)	Figure 8.9
10.0	32 – 4	25	3.28:1 (5.94:1)	Figure 8.10
1.0	8 – 4	25	2.43:1 (5.19:1)	Figure 8.11
1.0	32 – 4	50	4.62:1 (9.81:1)	Figure 8.12
1.0	32 – 4	80	5.92:1 (14.55:1)	Figure 8.13
1.0	32 – 4	125	7.49:1 (22.17:1)	Figure 8.14
1.0	32 – 4	200	25.32:1 (45.81:1)	Figure 8.15

Table 8.2: Summary of Test Results With Varying Parameters (sunset)

Some general observations are clear from these results. High maximum a values, such as in Figure 8.10, lead to divergence chromatically, as expected. The most interesting parameter is the MSE cutoff, which determines the quality (and compression ratio) of the final image. This is visible in the progression of sunset results (Figures 8.9, 8.5, 8.12, 8.13, 8.14, 8.15).

Natural images, such as the sunset (Figure 8.1) and chapel (Figure 8.3) test images, tend to compress relatively well with fractal image compression. Artificial images, such as textl (Figure 8.2), are less suitable; the compression ratios are lower and the distortion in the output more objectionable.

The visible figments of compression are blockiness and somewhat ragged edges in some cases. Continuous tones and textures are preserved fairly well, as are large-scale sharp transitions.

The resolution independence of fractal image compression may be seen from Figure 8.16, which is Figure 8.5 decompressed at twice the normal resolution.

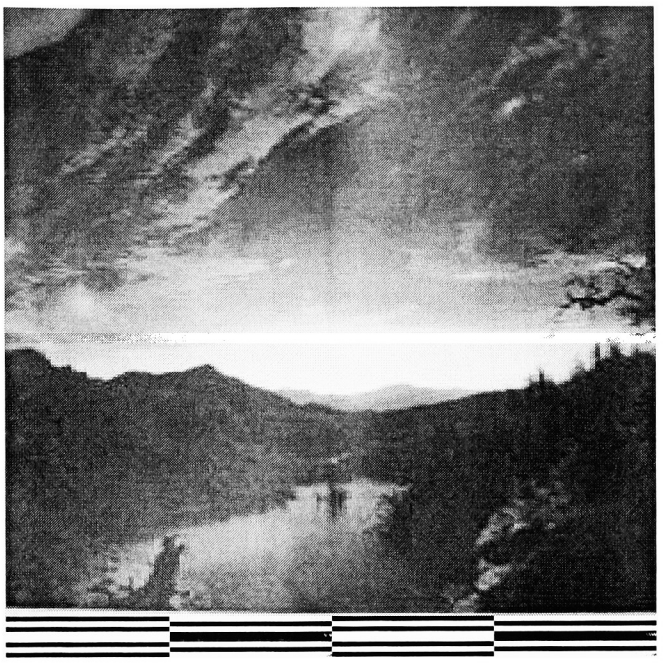


Figure 8.1: "sunset" Test Image (Original)

Although the alchemists were nc
and were clearly not in the intellectual
something that the philosophers had nc
various materials to prescribed treati
scribed as laboratory methods. These
laboratories, not only uncovered man
the systematic experimentation that is
Alchemy began to decline in th
1541), a Swiss physician and outspc
strongly advocated that the objectives
of medicine and the curing of human
ccnary efforts of alchemists to conver
But the real beginning of mode
during the Renaissance. Nicolaus Co
mer, succeeded in upsetting the genet

Figure 8.2: "textl" Test Image (Original)

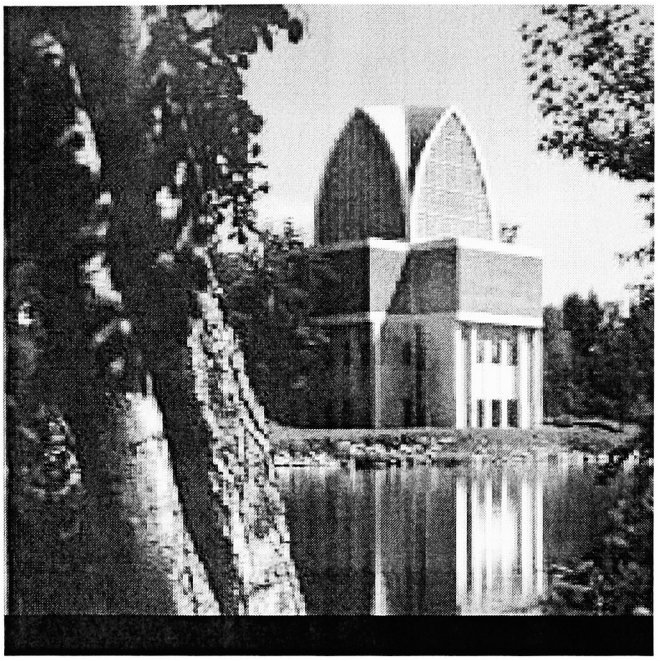


Figure 8.3: "chapel" Test Image (Original)

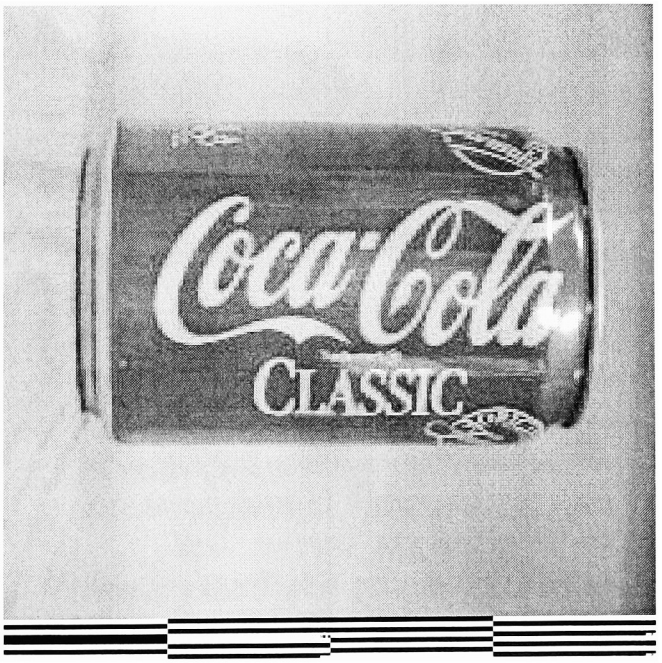


Figure 8.4: "coke" Test Image (Original)

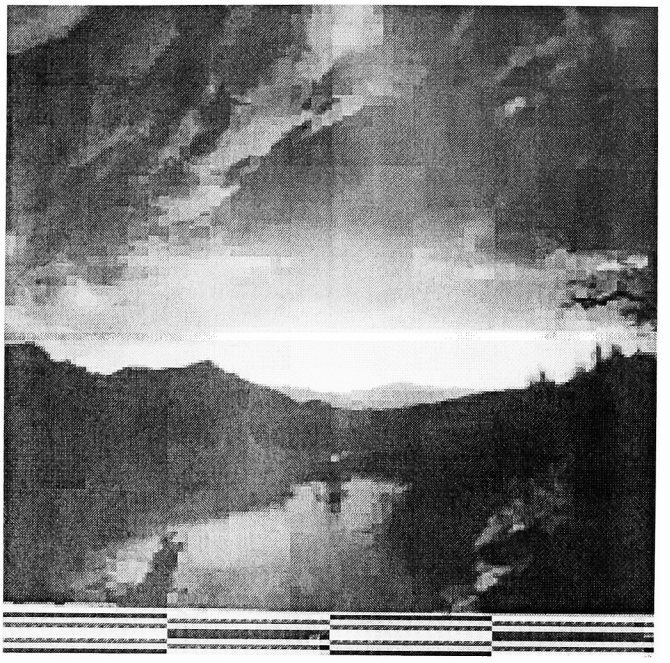


Figure 8.5: "sunset" Test Result

Although the alchemists were n
and were clearly not in the intellectual
something that the philosophers had n
various materials to prescribed treat
scribed as laboratory methods. These
laboratories, not only uncovered man
the systematic experimentation that is

Alchemy began to decline in th
1541), a Swiss physician and outspo
strongly advocated that the objectives
of medicine and the curing of human
cenary efforts of alchemists to conver

But the real beginning of mod
during the Renaissance. Nicolaus Co
mer, succeeded in upsetting the genet

Figure 8.6: "textl" Test Result

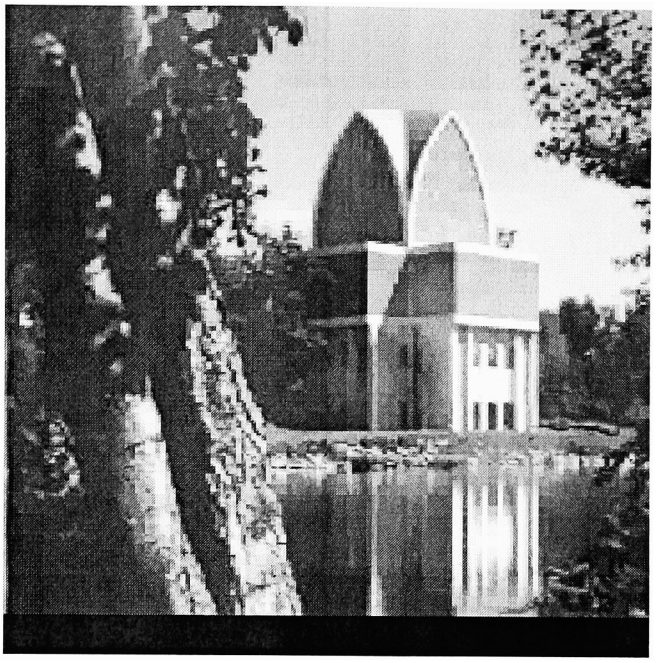


Figure 8.7: "chapel" Test Result

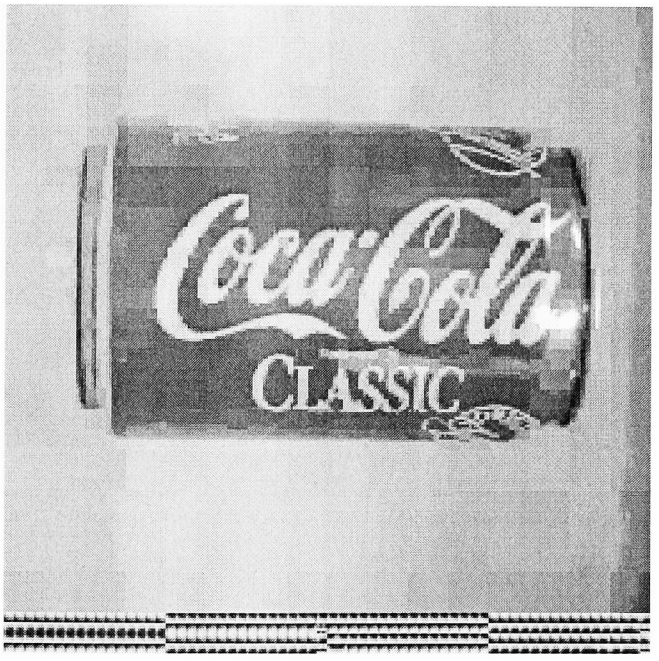


Figure 8.8: "coke" Test Result

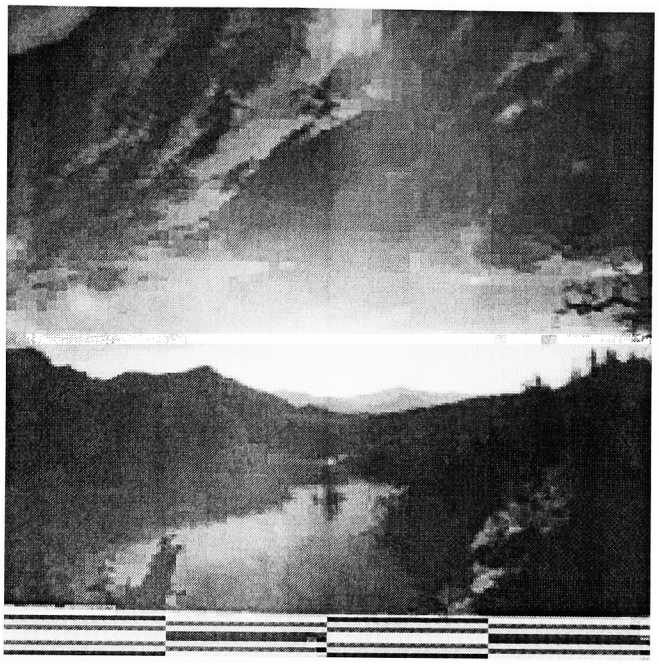


Figure 8.9: "sunset" Test Result (MSE cutoff = 10)

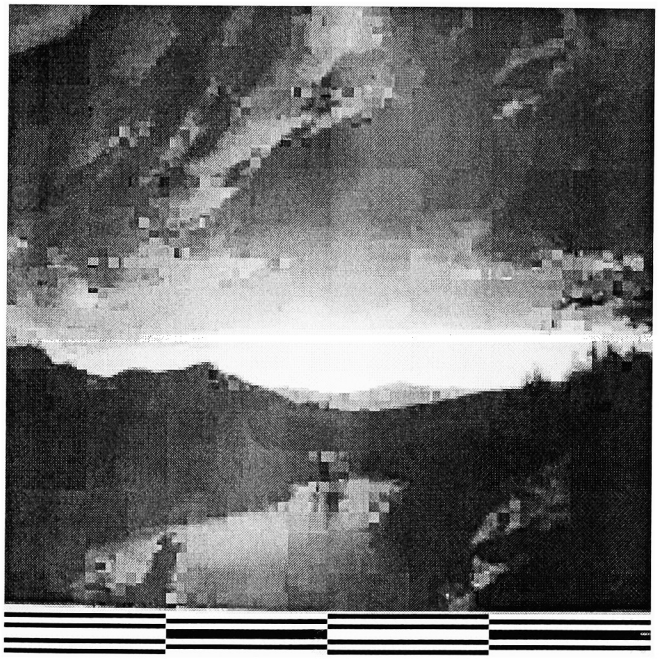


Figure 8.10: "sunset" Test Result (Max $a = 10$)

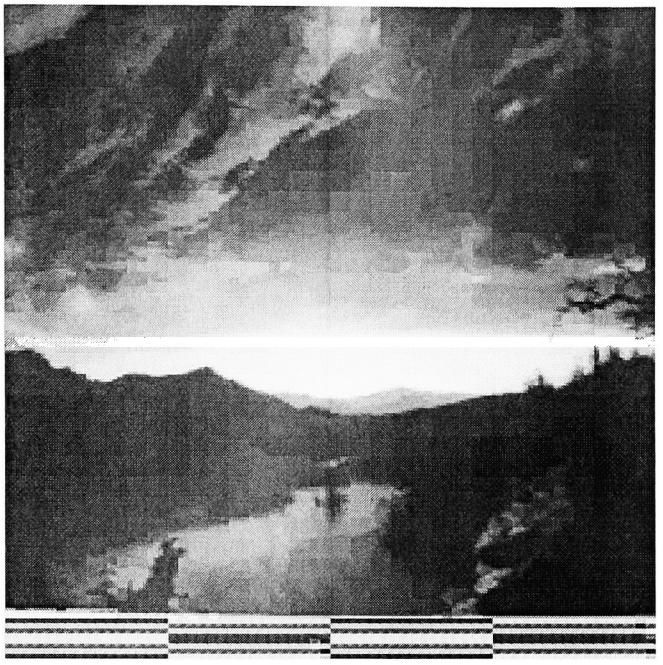


Figure 8.11: "sunset" Test Result (Initial Blocksize = 8)

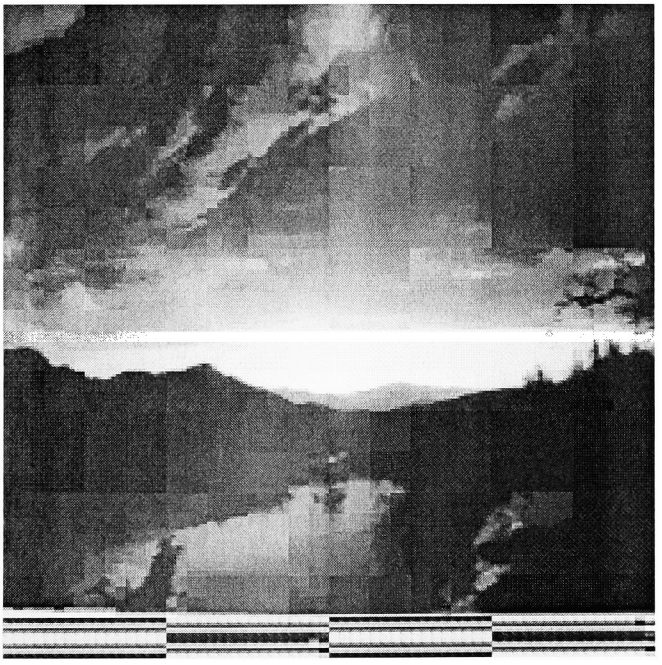


Figure 8.12: "sunset" Test Result (MSE cutoff = 50)

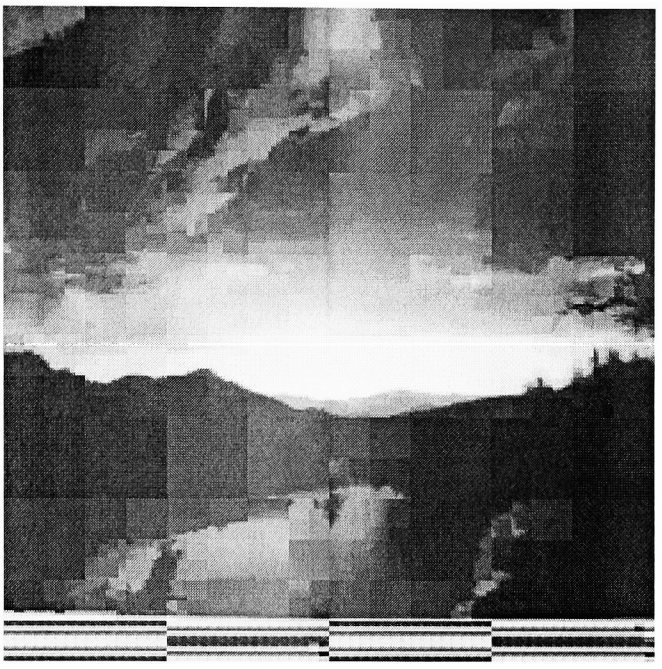


Figure 8.13: "sunset" Test Result (MSE cutoff = 80)

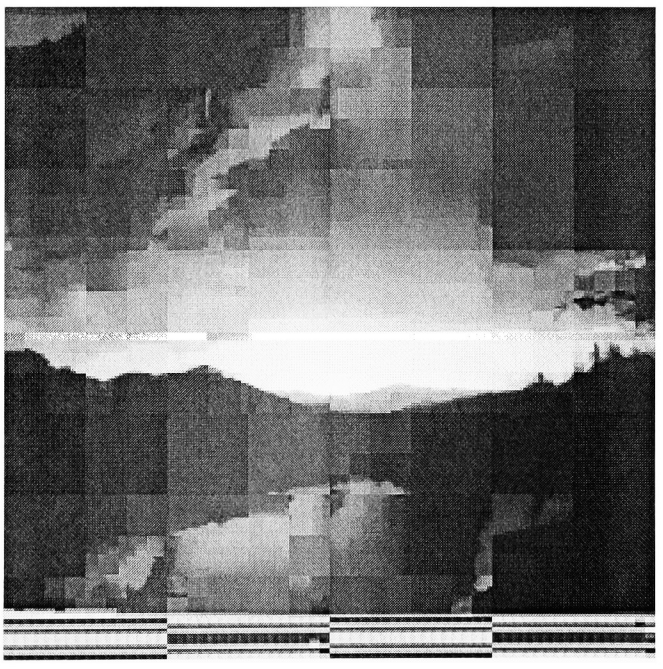


Figure 8.14: "sunset" Test Result (MSE cutoff = 125)

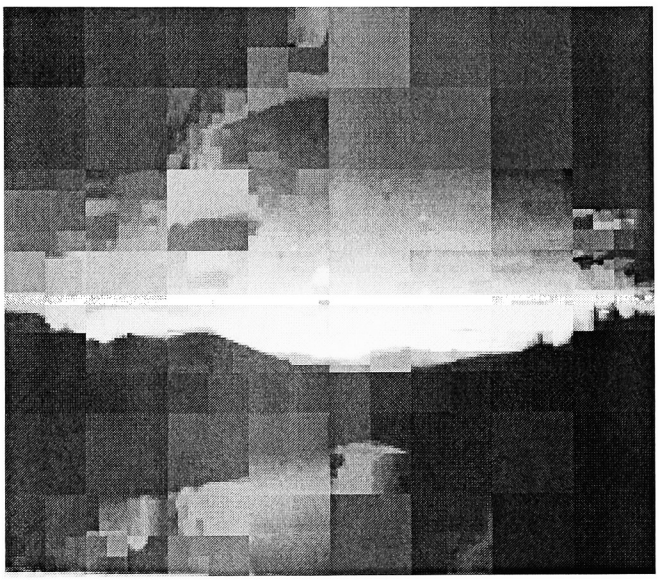


Figure 8.15: "sunset" Test Result (MSE cutoff = 200)

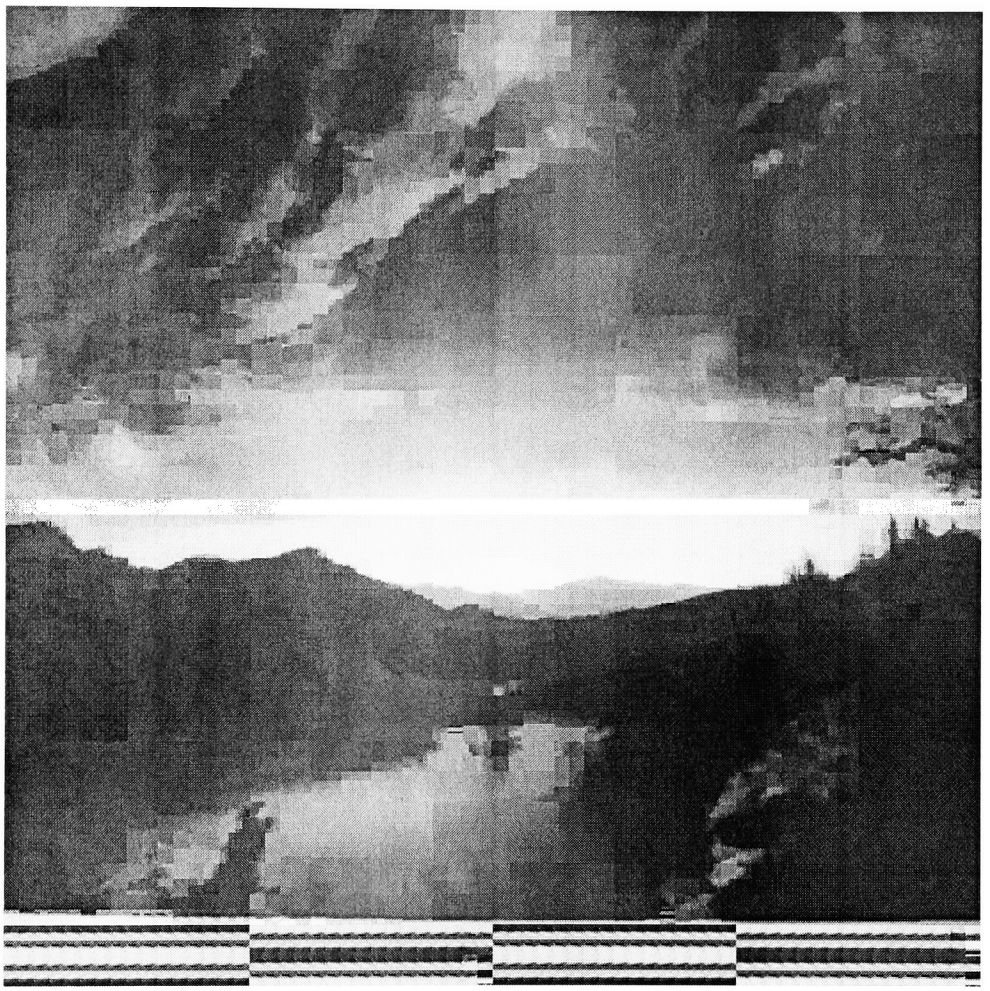


Figure 8.16: "sunset" Test Result (Magnification = 2)

8.2 Hardware Simulation

The sample image selected for hardware simulation is shown in Figure 8.17. This image was compressed using only 8×8 blocks; the results are shown in Figure 8.18. Compression using the software compressor, with similar parameters, resulted in Figure 8.19. The options used were `-a 1 -b 8 -c 1 -m 8 -h`; with these options, the results should be identical (except for possible miniscule differences due to floating point rounding errors). Comparing the transforms emitted shows that this is indeed the case; the only differences are insignificant variations in the a and b coefficients, which is attributable to slightly different rounding errors in the floating point computations producing these figures. (These values are computed by the driver program, not by the simulated hardware; the results found by the simulated hardware agree exactly with the predicted results generated by the compression software.)

Figure 8.17: Test Image for Hardware Simulation

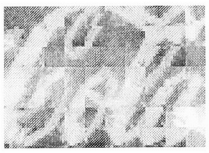


Figure 8.18: Result Image Generated by Hardware Simulation

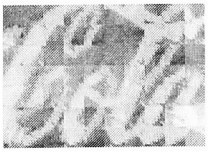


Figure 8.19: Result Image Generated by Compression Software

The simulation required approximately 200 clock cycles to load each range block and about 2,150,000 clock cycles to perform comparisons with all 2,560 domain blocks. (Each domain block comparison requires about 840 clock cycles for 8×8 blocks.) With a 10 MHz simulated clock, the total simulated time required for compression was about 220 ms. The time required for loading the range blocks

is insignificant when compared with the time required for the comparisons; both times grow linearly with the area of the image.

It is expected that an implementation using current technology would have a significantly faster clock speed than 10 MHz. This value was used for simulation because it allowed easy conversion between time and clock cycles when analyzing the results. With a 10 MHz clock and an adequate number of pipeline units, performing compression using 8×8 blocks on a 256×256 image would take a little over 3 seconds.

Chapter 9

Synthesis

9.1 Tools

The Synopsys Design Compiler was used to synthesize the design. A target architecture of LSI 10K was used. (The design is in no way tied to this target architecture; it is simply a reasonable placeholder.)

9.2 Procedures and Scripts

Appendix B contains a copy of the scripts used to synthesize the system. Various optimizations are used, particularly ungrouping certain portions of the design. All parts are synthesized assuming a 20 MHz clock.

The set_dont_use command near the start of several of the scripts exists to get around a difficulty with the particular library installation here; these specific instances require re-analysis, but the source code for them is missing.

Following standard practice, the `CLOCK` and `RESET` networks were not optimized at all. These global signals are, depending on the architecture, generally either routed semi-manually or constructed using special-purpose, pre-built circuitry.

9.3 Difficulties

The memory block in the range block chunk was the only problematic area for synthesis. A generic placeholder entity for this memory block is used in the design; it is expected that an actual implementation would replace the generic design with one specific to (and optimized for) the target architecture. Design Compiler is not especially adept at synthesizing memory blocks.

(Synopsys does provide some wrappers for architecture specific memories; unfortunately, none of these wrappers were for memories with an asynchronous read port, as the range block chunk was designed to use. Using a synchronous

memory would require significant changes to at least the range block chunk and potentially to other parts of the design.)

9.4 Results

The synthesis was performed for a system with four pipeline units. An actual implementation would contain as many of these units as practical, given the area constraints of the device; ideally, this number should be much higher than four.

Appendix C contains some sample schematics of the synthesized system. Only representative portions are included because the complete set of schematics would require several hundred pages.

Synthesis of the entire design from scratch (with nothing in the synthetic cache used by Synopsys) using these scripts takes about two or three hours on an HP 9000/785 system.

The areas reported by Design Compiler are in "gate equivalences." It keeps track of combinational area and noncombinational (sequential) area separately.

Component	Cells Used	Comb. Area	Seq. Area	Total Area
Memory Interface Unit	2,935	5,697	7,294	12,991
Host Interface Unit	268	413	1,486	1,899
Pipeline Unit (No Memory)		6,280	14,524	20,804
Pipeline Unit (As Implemented)		55,896	71,868	127,764
MAC Chunk	143	217	853	1,070
Parameter Computation Chunk	2,002	4,244	7,561	11,805
Range Block Chunk		49,916	57,483	107,399
Memory Block	21,315	49,616	57,344	106,960
Other Logic		300	139	439
Total Area (No Memory)		31,230	66,876	98,106
Total Area (As Implemented)		229,694	296,252	525,946

Table 9.1: Summary of Synthesized Area

Table 9.4 contains a summary of the area statistics for the various components of the design. The cells (in the cells required) column refer to LSI 10K standard cell in the target library. A few components do not have this information because they contained references to other entities, either generated by Design Compiler or instantiated directly; such components are reported as a single cell.

The difficulty of Design Compiler in generating memory blocks should be immediately evident from this table. Over four-fifths of the area required by the design is consumed by these (fairly small) memory blocks. If a target-specific memory design is not significantly better, it may be worthwhile to reduce the maximum block size.

The area required for the host interface will vary somewhat depending on the number of pipeline units implemented. Since the host interface is relatively small

and simple, this change should be insignificant when compared to the additional area required by the additional pipeline units.

Chapter 10

Conclusions and Extensions

In general, the design performed admirably in the testing. Actual hardware implementation would probably be best performed with an ASIC design (rather than some programmable logic system) due to the complexity of the design.

10.1 Possible Improvements and Extensions

10.1.1 Memory Interface Unit

Cache and Memory Interface

As was mentioned in Section 5.1, portions of the memory interface require modification for actual use based on the precise nature of the memory system. Part of this is designing a suitable cache, such as the very simple one developed here.

Larger caches would decrease the memory bandwidth requirements substantially; how necessary this is depends on the relative speed of the system to the memory bus and on what other devices share the bus. A few cache size possibilities are evident.

The smallest reasonable size, as is used here, is two machine words. This reduces the number of memory accesses required by two over having no cache (as a word is not accessed twice, once for each byte, in close succession).

A second possibility, useful primarily if the memory bus operates in a burst mode, is to have enough cache to store two lines of a domain block; this would enable the system to read memory using only burst mode transfers. While this does not change the number of memory locations accessed, it does typically speed up the accesses substantially over reading words piecemeal.

A third possibility is having enough cache to hold an entire domain block; this would allow most comparisons (all except for those at the start of a line in the image) to read only one column per block rather than the entire block. This reduces the memory bandwidth by a factor of the size of the blocks being used over the original scheme. (For example, it would take about $\frac{1}{8}$ of the current memory bandwidth for 8×8 blocks.)

A fourth possibility is to have enough cache to hold an entire row of domain blocks. This is adequate to ensure that the image is only read once during the course of a set of comparisons, yielding an average of not much more than one byte read per block compared.

In all cases, using a simple FIFO algorithm should yield fairly good cache performance, as the memory addressing is largely linear. (Larger caches, of course, would probably be set associative rather than the fully associative cache used in the current design.)

Denominator Computation Chunk

The denominator computation chunk currently starts comparisons in cases where the denominator is zero. Since division is nowhere actually performed by the hardware, this does not lead to illegal arithmetical operations. Testing also did not reveal any incorrect mappings from this strategy; the test images were compressed properly, even with such blocks.

It may be better not to perform comparisons in these cases; at the least, it may save some time with small blocksizes, where the parameter computation is the bottleneck on performance. It is easy to test, in software, whether or not the block is reasonably represented by a shade block.

10.1.2 Pipeline Unit

The parameter computation units currently use roughly one hundred clock cycles per block per isometry. This number could be reduced somewhat for smaller blocks without changing the overall design, since the maximum possible sums are smaller. If the number of cycles used were dynamically determined from the blocksize, the system would be a little faster for small blocks. (For large blocks, larger than approximately 10×10, the parameter computation unit processing time is not a bottleneck.)

It may also be helpful to determine if limiting the precision of the calculations would result in substantially worse performance, as was done in [12]. If similar results are applicable to this architecture, the time required for parameter computation could be reduced by a significant fraction.

10.2 Concluding Remarks

The design performed well in simulations and was shown to be synthesizable. A speedup of roughly 1000 times versus a pure software approach was predicted for similar implementation technologies.

Manufacture of the design developed here may be covered by patents owned by Iterated Systems, Inc. [13]; a reader wishing to produce this design or derivative designs should contact them. The author has no affiliation with Iterated Systems, nor has this design been specifically based on any design developed by them.

Appendix A

Software Implementation of Compression and Decompression

A.1 Introduction

The implementation is written in ANSI C on Unix (DEC OSF/1). The only operating system dependent sections involve file I/O and the call to `setpriority` in the compression program. (This call reduces the priority of the process to prevent it from causing angst among interactive users of the system. It could be removed without affecting the operation of the program.)

There are three pieces of code: the compression program, the decompression program, and a simple library to read and write MS Windows .BMP files.

A.1.1 Invocation

fcomp3

The command line arguments for fcomp3, the compression program, are listed in Table A.1.1. Options may be entered in any order.

The input file must be a grayscale MS Windows .BMP file. The two blocksize parameters are further constrained either to be an even power of two or to be three times an even power of two. (This is to ensure that quadtree dissection works accurately.)

If the maximum a parameter is greater than one it is possible that the LIFSM produced may be divergent chromatically. Whether or not it is divergent (or divergent in certain areas) depends on the input image and the other compression factors; often, a maximum a modestly greater than 1.0 will still produce a globally convergent LIFSM.

A small MSE cutoff leads to an accurate reconstruction of the original image and a relatively low compression ratio. A high MSE cutoff implies a high compression ratio (but a potentially poor reconstruction).

Compression takes a considerable amount of time, which varies quadratically with the number of pixels in the image. The test images used (256×256 pixels)

Option	Description	Default	Legal Range
-a	Maximum permitted a regression value	1.0	$0 < a < 256.7$
-b	Initial Blocksize	32	$2 \leq m \leq b \leq 96$
-c	MSE cutoff	25.0	$0 < c < 1315$
-h	Enable Hardware Emulation Mode	disabled	
-i	Input filename	none	
-m	Minimum Blocksize	3	$2 \leq m \leq b \leq 96$
-o	Output Filename	none	
-v	Enable Verbose Output	disabled	

Table A.1: fcomp3 Command Line Arguments

typically take around 15 minutes of cpu time to compress, depending on the image and the specific options used. (This is on a 532 MHz Alpha 21164A system.)

The hardware emulation mode simplifies the algorithm somewhat to better simulate what the hardware design does. (The change affects situations where linear regression produces a a value which is beyond the allowable range. With hardware emulation enabled, the mapping is not considered; with it disabled, the mapping is considered with the a parameter clamped to the maximum or minimum permitted value.)

fdecomp3

The command line arguments for the decompression program `fdecomp3` are listed in Table A.1.1. These may be entered in any desired order.

Option	Description	Default	Legal Range
-i	Input filename	none	
-m	Magnification	1	$1 \leq m$
-o	Output Filename	none	
-r	Number of Repetitions	20	$1 \leq r$
-v	Enable Verbose Output	disabled	

Table A.2: fdecomp3 Command Line Arguments

The input file must be one produced by the fcomp3 program; the output file will be a grayscale Windows .BMP file.

The number of repetitions required to converge on a (good approximation of) the attractor seems to be between ten and fifteen; much smaller numbers of repetitions lead to noticeably poorer images, and greater numbers do not lead to substantially different images.

Decompression is relatively quick, especially when compared with compression. Decompression which produces a large image may take a few seconds on a

fairly fast system.

A.2 Compression Program: fcomp3.c

```
/*****************************************************************************
 *                                                                           *
 *   f c o m p 3                                                             *
 *                                                                           *
 *  Fractally compress an image; various parameters of the compression are   *
 *  configurable, including the range of a values permitted, the sizes of    *
 *  the range blocks, and the quality (versus file size) desired.            *
 *                                                                           *
 *****************************************************************************/

#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>
#include "bmp.h"

/* Default values */
#define INITIAL_BLOCK_SIZE 32
#define INITIAL_MIN_BLOCK_SIZE 3
#define DEFAULT_CUTOFF 25
#define DEFAULT_MAXA 1

/* Access for mallocable two-dimensionable arrays.  This is a bit hard to do */
/* neatly in C; it only does statically defined multidimensional arrays */
/* using nice array syntax, AFAIK */

#define SOURCE(x, y) source[ (y)*srow + (x) ]
#define SAVG(x, y) savg[ (y)*avgrow + (x) ]
#define SSQR(x, y) ssqr[ (y)*avgrow + (x) ]
#define SUMY(x, y) sumy[ (y)*avgrow + (x) ]
#define SUMYY(x, y) sumyy[ (y)*avgrow + (x) ]

/* Other macros */
#define ABS(a) ((a)<0?-(a):(a))

/* The following are for write_bit */
#define ZERO_BIT 0
#define ONE_BIT 1
#define FLUSH_BITS -1

struct my_header {
    short blocksize;
    int width;
    int height;
};

struct filebit {    /* Stuff to get stored in output file */
                    /* These are shorts for ease; they could easily be characters or */
                    /* bitfields and thereby increase the compression ratio. */
    short dx;       /* The x coordinate */
    short dy;       /* The y coordinate */
    short orient;   /* The specific mapping */
    float a;        /* a greymap factor */
    float b;        /* b greymap factor */
};

struct qt_elem {    /* Element in (internal) quadtree */
                    /* Block size -- unneeded */
    short rx;       /* Range x position */
    short ry;       /* Range y position */
```

```
    short orient;       /* Orientation */
    short dx;       /* Domain x */
    short dy;       /* Domain y */
    float a;        /* A graymap factor: y = ax + b */
    float b;        /* B graymap factor */
    float r;        /* Regression coefficient */
    float rs;       /* Saved rs param */
    float ds;       /* Saved ds param */
    struct qt_elem *next;   /* Next in (breadth-first) list do */
    struct qt_elem *quad1;  /* Quadtree kid 1 */
    struct qt_elem *quad2;  /* Quadtree kid 2 */
    struct qt_elem *quad3;  /* Quadtree kid 3 */
    struct qt_elem *quad4;  /* Quadtree kid 4 */
};

struct qt_elem *top;
struct qt_elem *currlist;
struct qt_elem *nextlist;
struct qt_elem *nextlistend;

unsigned int *source;       /* Source data array */
unsigned int *savg;         /* Averages of source data pixels */
unsigned int *ssqr;         /* Squares of source data pixels */
unsigned long *sumy;        /* Sums of domain blocks */
unsigned long *sumyy;       /* Sums of squares of domain blocks */

int srow;       /* xsize - xcrop */
int avgrow;     /* xsize - xcrop - 1 */
int scol;       /* ysize - ycrop */
int avgcol;     /* ysize - ycrop - 1 */
int sumrow;     /* xsize - xcrop - blocksize, kind of */
int sumcol;

int xsize, ysize;   /* Size of original picture */
int xcrop, ycrop;   /* Number of pixels chopped off edge */
int rowlen;         /* Number of bytes in a row */
unsigned long blockpixs = INITIAL_BLOCK_SIZE * INITIAL_BLOCK_SIZE;
                    /* size of block squared */
int dblocklen = 2 * (INITIAL_BLOCK_SIZE - 1);   /* 2*(blocksize - 1) */
struct my_header hdr;

int blocksize = INITIAL_BLOCK_SIZE; /* Size of a range block */
int minblocksize = INITIAL_MIN_BLOCK_SIZE;  /* Minimum size of a range block */
int verbose;                        /* Verbose flag */
int hwflag;                         /* Hardware emmulation flag */
float cutoff = DEFAULT_CUTOFF;      /* closeness cutoff for r squared */
float maxa = DEFAULT_MAXA;          /* maximum a parameter */
char *infile = NULL;                /* Input file name */
char *outfile = NULL;               /* output file name */

struct bmpfile *inbmp;              /* Input bitmap file */
FILE *infilep;                      /* Input file pointer */
int outfilenum;                     /* Output file descriptor */

void precompute_stuff();            /* Once per quadtree level init */
void do_args( int argc, char **argv );
void initialize();                  /* Once per execution init */
void do_block( struct qt_elem *block );
void *my_malloc( unsigned long size );
int check_and_lower_tree();
void write_qtmap();
void write_transform( struct qt_elem *which );
void write_bit( int bitval );
void write_qtmap_elem( int blocksize, struct qt_elem *here );

int main( int argc, char ** argv ) {
    int err;
```

```
inbmp = 0;

/* Play with command line */

do_args( argc, argv );

/* We have our options.  We need to do a few things: */
/* o read in the file */
/* o Set stuff up for compression */
/* o For each quadtree level: */
/*   x precompute domain stuff (for speed) */
/*   x Do the compression */
/* o Write out the resultant file. */

/* Read in the file */
infilep = fopen( infile, "r" );
if( infilep == NULL ) {
    fprintf( stderr, "Error: input file not openable!\n" );
    exit( -1 );
}

inbmp = read_bmp( infilep );
err = cleanup_bmp( inbmp );
if( err != 0 ) {
    fprintf( stderr, "Error cleaning up bitmap!\n" );
    exit( -1 );
}

/* Determine a few important constants and such */

xsize = inbmp->bmi->bmih.biWidth;
ysize = inbmp->bmi->bmih.biHeight;
xcrop = xsize % blocksize;  /* This should really be changed... */
ycrop = ysize % blocksize;  /* but dealing with partial cols/rows is hard. */
rowlen = ROWLEN( inbmp );   /* Bytes in bmp row */

/* Report vital stats to user if appropriate */
if( verbose ) {
    printf( "******** Invocation details *********\n\n" );
    printf( "Source image:   %s\n", infile );
    printf( "Image size:     %d x %d\n", xsize, ysize );
    printf( "Row length:     %d\n", rowlen );
    printf( "Initial Block:  %dx%d\n", blocksize, blocksize );
    printf( "Minimum Block:  %dx%d\n", minblocksize, minblocksize );
    printf( "MSE Cutoff:     %f\n", cutoff );
    printf( "Maximum A:      %f\n", maxa );
    printf( "************************************\n\n" );
}

/* Report important stuff (errors, warnings) to user */

if( blocksize * 2 > xsize || blocksize * 2 > ysize ) {
    fprintf( stderr, "Error:  Block size too big for picture.\n" );
    exit( -1 );
}

if( xcrop != 0 && ycrop != 0 ) {
    fprintf( stderr, "Notice:  I shall crop %d pixels horizontally and "
            "%d vertically.\n", xcrop, ycrop );
} else {
    if( xcrop != 0 ) {
        fprintf( stderr, "Notice:  I shall crop %d pixels horizontally.\n",
                xcrop );
    }
    if( ycrop != 0 ) {
        fprintf( stderr, "Notice:  I shall crop %d pixels vertically.\n",
                ycrop );
    }
```

```
}

/* Be nice to your friends; this does the same as a nice 10 on UNIX. */
/* On other OSs, this could be replaced with something appropriate or */
/* just eliminated.  (This also improves the chances of the admins not */
/* killing off the program when they see it using a lot of CPU time; */
/* at least it doesn't steal the CPU from people doing "useful" work.) */

setpriority( PRIO_PROCESS, 0, 10 );

/* Precompute some things */

initialize();

/* Store important stuff for file header */
/* Done now because blocksize will change as time goes on. */

hdr.blocksize = blocksize;
hdr.width = xsize - xcrop;
hdr.height = ysize - ycrop;

/* A word on data structures used: */
/* The blocks at a single level are linked in a list to allow for easy */
/* breadth-first access (by following the list).  The descendents of a */
/* top-level block are linked in a tree.  At each level, nextlist is */
/* created to hold dissected blocks from thislist; after a level is */
/* done, nextlist becomes thislist.  top always points to the list at */
/* the very top of the tree, the initial block size. */

top = nextlist;     /* Set top to be the top of the quadtree */

/* Do the compression */

while( nextlist != 0 ) {
    if( verbose ) {
        printf( "--------> Starting with %dx%d blocks!\n",
                blocksize, blocksize );
    }
    currlist = nextlist;
    nextlist = nextlistend = 0;
    precompute_stuff();
    while( currlist != 0 ) {
        do_block( currlist );
        currlist = currlist -> next;
    }
    blocksize >>= 1;
    dblocklen = 2 * (blocksize - 1 );
    blockpixs = blocksize*blocksize;
    if( blocksize == 1 && nextlist != 0 ) {
        fprintf( stderr, "Error: could not compress--image too random!\n");
        exit( -1 );
    }
}

/* Write the output file */

/* Lower the height of the quadtree until we find the largest block */
/* which has an actual transform in it (is not split up.)   This makes */
/* the map a bit smaller, especially in cases where an obscenely huge */
/* initial blocksize was used (for the image/MSE combination)--such as */
/* a blocksize of 64 and a MSE of 4 or some such. */

if (verbose ) printf( "\n\n Checking and lowering tree...\n" );
while( check_and_lower_tree() ) {
    /* Intentionally empty */
}

/* Write the header */
```

```
    outfilenum = open( outfile, O_WRONLY|O_CREAT|O_TRUNC, 0644 );
    if( verbose) printf( "Writing header...\n" );
    write( outfilenum, &hdr, sizeof( struct my_header ) );

    /* Write the quadtree map (recursively) */
    if( verbose) printf( "Writing quadtree map...\n" );
    write_qtmap();

    /* (recursively) write the transforms out */
    if( verbose ) printf( "Writing transforms...\n" );
    currlist = top;
    while( currlist != 0 ) {
        write_transform( currlist );
        currlist = currlist -> next;
    }

    close( outfilenum );
    fprintf( stderr, "All done!\n" );
}

/* Here, we do comparisons for a single block.  This takes up the lion's */
/* share of the processor time, so it is more or less optomized for speed */
/* (on the system I've been using, at least--an alpha, where floating */
/* point operations seem nearly as quick as integer operations. */

void do_block( struct qt_elem *block ) {
    int rx, ry;
    int tempx, tempy;
    int dx, dy;
    float sumx, sumxx;
    float r;
    float a;
    float b;
    float den;
    float sumxxyy;
    unsigned long sumxy[8];     /* Results of doing the various transforms */
    int i, slow, fast;

    if( verbose ) {
        printf( "Doing %dx%d@(%d,%d)...",blocksize,blocksize,
                block->rx,block->ry );
        fflush( stdout );
    }

    block->r = 0.0;
    block->a = maxa;
    rx = block->rx;
    ry = block->ry;
    sumx = 0;
    sumxx = 0;
    for( tempx = 0; tempx < blocksize; tempx++ ) {
        for( tempy = 0; tempy < blocksize; tempy++ ) {
            sumx += SOURCE(rx+tempx, ry+tempy);
            sumxx += SOURCE(rx+tempx, ry+tempy)*
                SOURCE(rx+tempx,ry+tempy);
        }
    }

    /* Assume a shade block, compute r appropriately */
    /* (r = MSE; rs and ds are saved_r and saved_den as used by the */
    /* hardware) */

    block->dx = 0;
    block->dy = 0;
    block->orient = 0;
    block->a = 0;
    block->b = (float) sumx / (float) blockpixs;
    block->r = (((float)blockpixs) * sumxx - (sumx*sumx))/
```

```
        (float)(blockpixs*blockpixs);

if( block->r >= cutoff ) {
    /* Only do if shade is not good enough. */
    block->rs = 1.0;
    block->ds = 0.0;

    for( dx = 0; dx <= sumrow; dx++ ) {
        for( dy = 0; dy <= sumcol; dy++ ) {
            if( blockpixs * SUMYY(dx, dy) != SUMY(dx,dy)*SUMY( dx,dy ) ) {

                for( fast = 0; fast < 8; fast++ ) {
                    sumxy[fast] = 0;
                }
                for( slow = 0; slow < blocksize; slow++ ) {
                    for( fast = 0; fast < blocksize; fast++ ) {
                        /*orient    range and domain */
                        sumxy[0] += SOURCE(rx+slow,ry+fast) *
                                SAVG(dx+slow*2,dy+fast*2);
                        sumxy[1] += SOURCE(rx+fast,ry+slow) *
                                SAVG(dx+slow*2,dy+fast*2);
                        sumxy[2] += SOURCE(rx+slow,ry+fast) *
                                SAVG(dx+dblocklen-slow*2,dy+fast*2);
                        sumxy[3] += SOURCE(rx+fast,ry+slow) *
                                SAVG(dx+dblocklen-slow*2,dy+fast*2);
                        sumxy[4] += SOURCE(rx+slow,ry+fast) *
                                SAVG(dx+slow*2,dy+dblocklen-fast*2);
                        sumxy[5] += SOURCE(rx+fast,ry+slow) *
                                SAVG(dx+slow*2,dy+dblocklen-fast*2);
                        sumxy[6] += SOURCE(rx+slow,ry+fast) *
                                SAVG(dx+dblocklen-slow*2,dy+dblocklen-fast*2);
                        sumxy[7] += SOURCE(rx+fast,ry+slow) *
                                SAVG(dx+dblocklen-slow*2,dy+dblocklen-fast*2);
                    }
                }

                den = ((float)blockpixs)*SUMYY(dx,dy) -
                        ((float)SUMY(dx,dy))*SUMY(dx,dy);
                sumxxyy = sumx*sumx * ((float)SUMYY(dx,dy));

                for( i = 0; i < 8; i++ ) {

                    r = 2*((float)sumxy[i])*sumx*((float)SUMY(dx,dy));
                    r -= sumxxyy + ((float)blockpixs)*((float)sumxy[i])*sumxy[i];

                    if( r * block->ds < block->rs * den ) {
                        a = ((float) ((signed)(blockpixs)*(signed)sumxy[i]-
                                (signed)sumx*(signed)SUMY(dx,dy)))/
                                ((float)blockpixs*SUMYY(dx,dy)-
                                SUMY(dx,dy)*SUMY(dx,dy));

                        if( ABS(a) < maxa ) {
                            block->rs = r;
                            block->ds = den;
                            block->dx = dx;
                            block->dy = dy;
                            block->orient = i;
                            block->a = a;
                            block->b = (sumx - a*SUMY(dx,dy))/blockpixs;
                        } else if( !hwflag ) {
                            /* We may still do better than the best if */
                            /* we clamp a at the max or min value */
                            if( a < 0 ) {
                                a = -maxa;
                            } else {
                                a = maxa;
                            }
                            r = (float)( blockpixs ) *
```

```
                                (sumxx - a*2*sumxy[i] + a*a*SUMYY(dx,dy ) ) -
                                sumx*sumx + a*2*sumx*SUMY(dx,dy) -
                                a * a * ((float)SUMY(dx,dy)) * SUMY(dx,dy);
                            r /= blockpixs;
                            r -= sumxx;

                            if( r * block->ds < block->rs ) {
                                block->rs = r * den;
                                block->ds = den;
                                block->dx = dx;
                                block->dy = dy;
                                block->orient = i;
                                block->a = a;
                                block->b = ((float)( sumx - a*SUMY(dx,dy)))/blockpixs;
                            }
                        }
                    }
                    /* Previously, if the MSE was the same but the a */
                    /* value was better, we stored the new value; this */
                    /* helps the result converge a bit faster.  This is */
                    /* not done anymore because the hardware cannot */
                    /* easily do the same and because it's a bit harder */
                    /* to figure out with the rs/ds approach. */
                }
            }
        }
    }

    r = ((block->rs/block->ds)+sumxx)/(float)blockpixs;
    if( r < block->r ) {    /* What we found was better */
        block->r = r;       /* Update the MSE in the block */
    } else {
        /* Shade was better.  We do know it's not under the cutoff; */
        /* we still update this in case we are already at the min */
        /* blocksize the user has allowed.  It's an unlikely case, but */
        /* it can happen and we should do the right thing. */
        block->dx = 0;
        block->dy = 0;
        block->orient = 0;
        block->a = 0;
        block->b = (float) sumx / blockpixs;
        block->r = (float)(blockpixs * sumxx - sumx*sumx)/(float)(blockpixs*blockpixs);
    }
}
if( block->r > cutoff && blocksize > minblocksize ) {
    /* Perform a quadtree disection */
    if( verbose ) {
        printf( "  Dissecting...Best MSE=%f\n", block->r );
    }
    block->quad1 = (struct qt_elem *)my_malloc( sizeof( struct qt_elem ) );
    block->quad2 = (struct qt_elem *)my_malloc( sizeof( struct qt_elem ) );
    block->quad3 = (struct qt_elem *)my_malloc( sizeof( struct qt_elem ) );
    block->quad4 = (struct qt_elem *)my_malloc( sizeof( struct qt_elem ) );
    if( nextlist == 0 ) {
        nextlist = block->quad1;
    } else {
        nextlistend->next = block->quad1;
    }
    block->quad1->next = block->quad2;
    block->quad2->next = block->quad3;
    block->quad3->next = block->quad4;
    block->quad4->next = 0;
    nextlistend = block->quad4;
    block->quad1->rx = block->quad3->rx = block->rx;
    block->quad1->ry = block->quad2->ry = block->ry;
    block->quad2->rx = block->quad4->rx = block->rx + (blocksize >> 1);
    block->quad3->ry = block->quad4->ry = block->ry + (blocksize >> 1);
} else {
```

```
        /* We found a winner!  Either the MSE was good enough, or we can't */
        /* get any smaller than we already have. */
        if( verbose ) {
             printf( "MSE=%f (%d, %d) a=%f b=%f o=%d rs=%f ds=%f\n",
                    block->r, block->dx, block->dy, block->a,
                    block->b, block->orient, block->rs, block->ds );
        }
    }
}

/* Do command line argument processing. */

void do_args( int argc, char **argv ) {
    extern char *optarg;
    extern int optind;
    int opterr;
    int chr;
    int err = 0;
    int temp;

    optind = 1;

    while(( chr = getopt( argc, argv, "a:b:c:hi:o:m:v" )) != EOF ) {
        switch( chr ) {
        case 'a':
            maxa = atof( optarg );
            if( maxa <= 0 || maxa >= 256.7 ) {
                fprintf(stderr, "Error: maximum A parameter "
                        "must be betwixt zero and two "
                        "hundred fifty-six and seven tenths.\n" );
                err++;
            }
            break;

         case 'm':
            minblocksize = atoi( optarg );
            if( minblocksize < 3 || minblocksize > 96 ) {
                fprintf(stderr, "Error: minblocksize must be "
                        "betwixt three and ninety-six.\n" );
                err++;
            }
            /* Test minblocksize for 2^n or 3*2^n */
            temp = minblocksize;
            while( (temp & 0x001 ) == 0 ) {
                temp >>= 1;
            }
            if( temp != 1 && temp != 3 ) {
                fprintf( stderr, "Error: minblocksize must be 2^n or 3*2^n.\n" );
                err++;
            }
            break;

        case 'b':
            blocksize = atoi( optarg );
            if( blocksize < 2 || blocksize > 96 ) {
                fprintf(stderr, "Error: blocksize must be "
                        "betwixt two and ninety-six.\n" );
                err++;
            }
            /* Test blocksize for 2^n or 3*2^n */
            temp = blocksize;
            while( (temp & 0x001 ) == 0 ) {
                temp >>= 1;
            }
            if( temp != 1 && temp != 3 ) {
                fprintf( stderr, "Error: blocksize must be 2^n or 3*2^n.\n" );
                err++;
            }
```

```
                blockpixs = blocksize * blocksize;
                dblocklen = 2*(blocksize - 1);
                break;

            case 'c':
                cutoff = atof( optarg ) ;
                if( cutoff < 0 || cutoff > 1315 ) {
                    fprintf( stderr, "Error: cutoff must be "
                            "betwixt zero and one thousand three hundred fifteen.\n" );
                    err++;
                }
                break;

            case 'i':
                infile = optarg;
                break;

            case 'o':
                outfile = optarg;
                break;

            case 'v':
                verbose++;
                break;

            case 'h':
                hwflag++;
                break;

            default:
                err++;
                break;
            }
        }
        if( infile == NULL ) {
            err++;
        }
        if( outfile == NULL ) {
            err++;
        }
        if( minblocksize > blocksize ) {
            fprintf( stderr, "Error:  minblocksize must be less "
                    "than blocksize.\n" );
            err++;
        }

        if( err != 0 ) {
            fprintf( stderr, "Usage: %s -i <infile.bmp> -o <outfile> "
                    "[other options]\n", argv[0] );
            fprintf( stderr, "Options: -a <max a>          "
                    "(default is 3.0)\n" );
            fprintf( stderr, "         -b <init blocksize> "
                    "(default is 32)\n" );
            fprintf( stderr, "         -c <r cutoff>       "
                    "(default is 1.0)\n" );
            fprintf( stderr, "         -h                  "
                    "Hardware modeling flag\n" );
            fprintf( stderr, "         -m <min blocksize>  "
                    "(default is 2)\n" );
            fprintf( stderr, "         -v                  "
                    "Verbose mode\n" );
            exit( err );
        }
        if( hwflag && maxa != 1 ) {
            fprintf( stderr, "Warning: hardware flag set and max a not 1; hardware "
                    "modeling will be inexact.\n" );
        }
}       }
```

```
/* Once-per-execution initialization stuff; mainly filling source, savg,
/* ssqr, allocating memory, and creating the initial list of blocks */

void initialize() {
    int dx;
    int dy;
    struct qt_elem *temp;

    /* Compute some constants which will be used throughout. */
    srow = xsize - xcrop;
    avgrow = xsize - xcrop - 1;
    scol = ysize - ycrop;
    avgcol = ysize - ycrop - 1;

    /* Allocate memory; this ought to use my_malloc, I guess. */
    source = (unsigned int *)malloc( sizeof( unsigned int ) * (srow+1)*(scol+1) );
    savg = (unsigned int *)malloc( sizeof( unsigned int ) * (avgrow+1)*(avgcol+1) );
    ssqr = (unsigned int *)malloc( sizeof( unsigned int ) * (avgrow+1)*(avgcol+1) );
    sumy = (unsigned long *)malloc( sizeof( unsigned long ) * (avgrow+1)*(avgcol+1) );
    sumyy = (unsigned long *)malloc( sizeof( unsigned long ) * (avgrow+1)*(avgcol+1) );

    if( source == NULL || savg == NULL || ssqr == NULL ||
            sumy == NULL || sumyy == NULL ) {
        fprintf( stderr, "Error: Out Of Memory!!!\n" );
        exit( -1 );
    }

    /* Fill source array */
    for( dx = 0; dx < srow; dx++ ) {
        for( dy = 0; dy < scol; dy++ ) {
            SOURCE( dx, dy ) = inbmp->data[dy*rowlen+dx];
        }
    }

    /* Compute averages, squares of averages */
    if( verbose ) {
        printf( "Computing averages and squares...\n" );
    }

    for( dx = 0; dx < avgrow; dx++ ) {
        for (dy = 0; dy < avgcol; dy++ ) {
            SAVG( dx, dy ) = ( SOURCE(dx, dy) + SOURCE(dx, dy+1) +
                    SOURCE(dx+1, dy) + SOURCE(dx+1, dy+1) +
                    2 ) / 4;
            if( SAVG(dx, dy) < 0 || SAVG(dx, dy) > 255 ) printf( "error\n" );
            SSQR(dx, dy) = SAVG(dx, dy)*SAVG(dx, dy);
        }
    }

    /* Set up initial list of blocks to do */
    /* This first block will be free'd and is just to prevent a special */
    /* case within the loop that follows. */
    nextlist = (struct qt_elem *)my_malloc( sizeof( struct qt_elem ) );
    nextlist->next = 0;
    nextlistend = nextlist;
    for( dx = 0; dx < xsize-xcrop; dx += blocksize ) {
        for( dy = 0; dy < ysize-ycrop; dy += blocksize ) {
            nextlistend->next = (struct qt_elem *)my_malloc( sizeof( struct qt_elem ) );
            nextlistend=nextlistend->next;
            nextlistend->rx = dx;
            nextlistend->ry = dy;
            nextlistend->r = 0.0;
            nextlistend->quad1 = 0;
            nextlistend->quad2 = 0;
            nextlistend->quad3 = 0;
            nextlistend->quad4 = 0;
        }
```

```
    }
    nextlistend->next = 0;

    /* Free that extra one at the start; currlist is used as a temp */
    currlist = nextlist;
    nextlist = nextlist->next;
    free( currlist );
    currlist = 0;

}

/* once-per-blocksize initialization--sumy and sumyy arrays */

void precompute_stuff() {
    int i;
    int dx;
    int dy;
    long sum;
    long sumsq;

    sumcol = ysize-ycrop- (blocksize*2);
    sumrow = xsize-xcrop- (blocksize*2);

    for( dx = 0; dx <= sumrow; dx++ ) {
        for( dy = 0; dy <= sumcol; dy++ ) {
            SUMY( dx, dy ) = 0;
            SUMYY( dx, dy ) = 0;
        }
    }

    /* Compute first four sums */

    if( verbose ) {
        printf( "Computing sums and sums of squares for %dx%d blocks...",
                blocksize, blocksize );
    }

    sum = 0;
    sumsq = 0;

    for( dx = 0; dx < blocksize * 2; dx+=2 ) {
        for ( dy = 0; dy < blocksize * 2; dy+=2 ) {
            sum += SAVG(dx, dy);
            sumsq += SSQR(dx, dy);
        }
    }
    if( sum < 0 || sumsq < 0 ) printf( "ERROR!\n" );
    SUMY(0, 0) = sum;
    SUMYY(0, 0) = sumsq;

    sum = 0;
    sumsq = 0;

    for( dx = 1; dx < blocksize * 2; dx+=2 ) {
        for ( dy = 0; dy < blocksize * 2; dy+=2 ) {
            sum += SAVG(dx, dy);
            sumsq += SSQR(dx, dy);
        }
    }
    if( sum < 0 || sumsq < 0 ) printf( "ERROR!\n" );
    SUMY(1, 0) = sum;
    SUMYY(1, 0) = sumsq;

    sum = 0;
    sumsq = 0;

    for( dx = 0; dx < blocksize * 2; dx+=2 ) {
        for ( dy = 1; dy < blocksize * 2; dy+=2 ) {
```

```
            sum += SAVG(dx, dy);
            sumsq += SSQR(dx, dy);
        }
    }
    if( sum < 0 || sumsq < 0 ) printf( "ERROR!\n" );
    SUMY(0, 1) = sum;
    SUMYY(0, 1) = sumsq;

    sum = 0;
    sumsq = 0;

    for( dx = 1; dx < blocksize * 2; dx+=2 ) {
        for ( dy = 1; dy < blocksize * 2; dy+=2 ) {
            sum += SAVG(dx, dy);
            sumsq += SSQR(dx, dy);
        }
    }
    if( sum < 0 || sumsq < 0 ) printf( "ERROR!\n" );
    SUMY(1, 1) = sum;
    SUMYY(1, 1) = sumsq;

    /* Compute the rest reasonably efficiently (moving the edges) */
    for( dx = 0; dx < sumrow; dx++ ) {
        if( dx > 1 ) {
            sum = SUMY(dx-2, 0);
            sumsq = SUMYY(dx-2, 0);
            for( dy = 0; dy < blocksize*2; dy+=2 ) {
                sum += (signed) ( SAVG(dx+(blocksize*2-2),dy) -
                          SAVG(dx-2,dy) );
                sumsq += (signed) (SSQR(dx+(blocksize*2-2),dy) -
                          SSQR(dx-2,dy));
            }
            SUMY(dx, 0) = sum;
            SUMYY(dx, 0) = sumsq;

            sum = SUMY(dx-2, 1);
            sumsq = SUMYY(dx-2, 1);
            for( dy = 1; dy < blocksize * 2; dy+=2 ) {
                sum += (signed) (SAVG(dx+(blocksize*2-2),dy) -
                          SAVG(dx-2, dy));
                sumsq += (signed) (SSQR(dx+(blocksize*2-2),dy) -
                          SSQR(dx-2,dy));
            }
            SUMY(dx, 1) = sum;
            SUMYY(dx, 1) = sumsq;
        }

        for( dy = 2; dy < sumcol; dy++ ) {
            sum = SUMY(dx, dy-2);
            sumsq = SUMYY(dx, dy-2);
            for( i = 0 ; i < blocksize*2; i += 2 ) {
                sum += (signed) (SAVG(dx+i,dy+blocksize*2-2) -
                        SAVG(dx+i,dy-2));
                sumsq += (signed) (SSQR(dx+i,dy+blocksize*2-2)-
                        SSQR(dx+i,dy-2));
            }
            SUMY(dx, dy) = sum;
            SUMYY(dx, dy) = sumsq;
        }
    }
    if( verbose ) {
        printf( "done\n" );
    }
}

        .
void *my_malloc( unsigned long size ) {
    void *result;
```

```
    result = calloc(1, size );
    if( result == 0 ) {
        fprintf( stderr, "Error: Out of Memory!\n" );
        exit( -2 );
    }
    return result;
}

/* Determine if we can knock off the top level of the tree, and do it */

int check_and_lower_tree() {
    struct qt_elem *this;
    struct qt_elem *prev;

    for( this = top; this != 0; this = this->next ) {
        if( this->quad1 == 0 ) {
            if( verbose ) printf( "Can't lower any more.\n" );
            return 0;   /* e.g. we can't prune any more. */
        } else {
            if( verbose ) {
                printf( "." );
                fflush( stdout );
            }
        }

    }

    /* done with check; now lower */

    if( verbose ) printf( "Lowering--removing all blocksize %d\n",
             hdr.blocksize );

    prev = top;
    top = top->quad1;

    while( prev != 0 ) {
        this = prev->next;
        free( prev );
        prev = this;
    }

    hdr.blocksize = hdr.blocksize >> 1;
    return 1;   /* Check again */
}

/* bit by bit output; for quadtree map */

void write_bit( int bitval ) {
    static unsigned char this_byte = 0;
    static int bits_in_char = 0;

    switch( bitval ) {
    case ZERO_BIT:
        this_byte >>= 1;
        this_byte &= 0x7F;   /* Clear high bit */
        bits_in_char++;
        break;
    case ONE_BIT:
        this_byte >>= 1;
        this_byte |= 0x80;  /* Set high bit */
        bits_in_char++;
        break;
    case FLUSH_BITS:
        if( bits_in_char > 0 ) {
            bits_in_char = 8;
        }
        break;
    default:
```

```
        fprintf(stderr, "Error: bad magic constant given to "
                "write_bit = %d!\n", bitval );
        exit( -1 );
        break;
    }

    if( bits_in_char >= 8 ) {
        write( outfilenum, &this_byte, 1 );
        bits_in_char = 0;
        this_byte = 0;
    }
}

/* Spit out a transformation (recursively). */

void write_transform( struct qt_elem *this ) {
    struct filebit thisbit;
    static int i = 0;

    if( this->quad1 == 0 ) {
        thisbit.dx = this->dx;
        thisbit.dy = this->dy;
        thisbit.a = this->a;
        thisbit.b = this->b;
        thisbit.orient = this->orient;
        if( verbose ) printf( "T#%d  dx=%d dy=%d rx=%d ry=%d orient=%d a=%f b=%f\n",
                i++, (int) this->dx, (int) this->dy, (int) this->rx,
                (int) this->ry, (int) this->orient, this->a, this->b );
        write( outfilenum, &thisbit, sizeof( struct filebit ) );
    } else {
        write_transform( this->quad1 );
        write_transform( this->quad2 );
        write_transform( this->quad3 );
        write_transform( this->quad4 );
    }
}

/* Spit out a quadtree map (recursively) for the given element */

void write_qtmap_elem( int size, struct qt_elem *here ) {
    /* No need to write out map for base cases */

    if( size <= 3 ) {
        return;
    }

    if( here->quad1 == 0 ) {
        write_bit( ZERO_BIT );

    } else {
        write_bit( ONE_BIT );
        write_qtmap_elem( size >> 1, here->quad1 );
        write_qtmap_elem( size >> 1, here->quad2 );
        write_qtmap_elem( size >> 1, here->quad3 );
        write_qtmap_elem( size >> 1, here->quad4 );
    }
}

/* Spit out the entire quadtree map and flush the last (partially filled) */
/* byte at the end. */

void write_qtmap() {
    struct qt_elem *here;
    int blocksize = hdr.blocksize;

    for( here = top; here != 0; here = here -> next ) {
        write_qtmap_elem( blocksize, here );
    }
```

```
    write_bit( FLUSH_BITS );
}
```

A.3 Decompression Program: fdecomp3.c

```
/*****************************************************************************
 *                                                                           *
 *   f d e c o m p 3                                                         *
 *                                                                           *
 *   Decompress an image fractally compressed using fcomp3 or ftrans3; the   *
 *   image may be magnified in the process if desired.                       *
 *                                                                           *
 *****************************************************************************/

#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include "bmp.h"

/* The following ensures all array accesses are within the array; this is a */
/* good idea since we don't verify that the input file is okay in this */
/* regard and since we may have other strange bugs in the code. */

#if 1
#define PICT( x, y ) (pictdata[(((y)%ysize)*xsize) + ((x)%xsize)])
#define PICT2( x, y ) (pictdata2[(((y)%ysize)*xsize) + ((x)%xsize)])
#else
#define PICT( x, y ) (*(pictdata + (y)*xsize + (x)))
#define PICT2( x, y ) (*(pictdata2 + (y)*xsize + (x)))
#endif

#define ONE_BIT 1
#define ZERO_BIT 0

struct filebit {    /* Stuff to get stored in output file */
                    /* This should probably contain bitfields and */
                    /* thereby increase the compression ratio. */
    short dx;       /* The x coordinate */
    short dy;       /* The y coordinate */
    short orient;   /* The specific mapping */
    float a;        /* a greymap factor */
    float b;        /* b greymap factor */
};

struct transform {
    int rx;         /* range x coord */
    int ry;         /* range y coord */
    int dx;         /* domain x coord */
    int dy;         /* domain y coord */
    int orient;     /* Orientation index */
    int blocksize;  /* Size of this block */
    float a;        /* A graymap factor */
    float b;        /* B graymap factor */
    struct transform *next;     /* Next one */
};

struct my_header {
    short blocksize;
    int width;
    int height;
};
```

```
char *infile;       /* Input file name */
char *outfile;      /* output file name */

int infilenum;      /* Input file descriptor */
FILE *outfilep;     /* Output file pointer */

/* We keep two copies of the image around.  We apply the transforms to one */
/* to produce a new generation in the other, then swap the pointers around */
/* and repeat. */

float *pictdata;
float *pictdata2;
int xsize;
int ysize;
int maxblocksize;
int mag;
int iterations;
int verbose;
struct transform *top;
struct transform *last; /*For building the list up */

void apply_transform( struct transform *which );
void read_transform( struct transform *which);
void read_qt_map( int x, int y, int blocksize );
int read_bit();

int main( int argc, char ** argv ) {
    int x;
    int y;
    int i;
    float *temp;
    struct bmpfile *bmp;
    extern char *optarg;
    extern int opterr;
    extern int optind;
    struct my_header hdr;
    int chr;
    int error = 0;
    unsigned char *datap;
    struct transform *this;


    infile = NULL;
    outfile = NULL;
    mag = 1;
    iterations = 20;
    optind = 1;
    verbose = 0;

    while((chr = getopt( argc, argv, "i:o:m:r:vV" )) != EOF ) {
        switch( chr ) {
        case 'i':
            infile = optarg;
            break;

        case 'o':
            outfile = optarg;
            break;

        case 'm':
            mag = atoi( optarg );
            if( mag<= 0 || mag > 50 ) error++;
            break;

        case 'r':
            iterations = atoi( optarg );
            if( iterations < 1 || iterations > 1000 ) error++;
            break;
```

```
    case 'v':   /* Allows for a very verbose output by specifying both */
    case 'V':   /* lowercase and uppercase v */
        verbose++;
        break;

    default:
        error++;
        break;
    }
}

if( error > 1 || infile == NULL || outfile == NULL ) {
    fprintf( stderr, "Usage:  %s -i <infile> -o <outfile> "
            "[options...]\n", argv[0] );
    fprintf( stderr, "Options: -m <magnification>        "
            "(Default is 1)\n" );
    fprintf( stderr, "         -r <repetitions>          "
            "(Default is 20)\n" );
    fprintf( stderr, "         -v                        "
            "Verbose mode\n" );
    exit( error );
}

/* Read in compressed info */

if( verbose ) {
    printf( "Reading in compressed file\n" );
}

infilenum = open( infile, O_RDONLY );
if( infilenum < 0 ) {
    fprintf( stderr, "Error: file '%s' unreadable.\n", infile );
    exit( 1 );
}

/* We assume the data is out there.  That probably isn't good. */

read( infilenum, &hdr, sizeof( struct my_header ) );

/* Read quadtree map and create transforms list */
top = (struct transform *)malloc( sizeof( struct transform ) );
if( top == 0 ){
    fprintf( stderr, "Error: out of memory!\n" );
    exit( 1 );
}
top->next = 0;
last = top;

xsize = hdr.width;
ysize = hdr.height;

for( x = 0; x < xsize; x += hdr.blocksize ) {
    for( y = 0; y < ysize; y+= hdr.blocksize ) {
        read_qt_map( x, y, hdr.blocksize );
    }
}

last->next = 0;     /* Null-terminate list */
this = top;         /* Get rid of first fake element */
top = top->next;
free( this );
this = 0;

/* Read in transforms */
for( this=top; this != 0; this = this->next ) {
    read_transform( this );
}
```

```
close( infilenum );

/* Adjust things for magnification */

if( mag != 1 ) {
    xsize *= mag;
    ysize *= mag;
    maxblocksize = hdr.blocksize * mag;
    for( this = top; this != 0; this = this->next ) {
        this->rx *= mag;
        this->ry *= mag;
        this->dx *= mag;
        this->dy *= mag;
        this->blocksize *= mag;
    }
}

/* Now, we can forget about the magnification; it's just as though the */
/* magnified picture were compressed initially. */

/* Picture computations are done in floating point to keep this easy. */
/* Picture data is converted to bytes only for output.  This approach */
/* is not a good idea for really big pictures, as it wastes a fair bit */
/* of memory in that case; it's also bad if floating point operations */
/* are especially slow (e.g. no FPU).  Neither are true for me now. */

pictdata = (float *) malloc( sizeof( float ) * xsize * ysize );
pictdata2 = (float *) malloc( sizeof( float ) * xsize * ysize );

if( verbose) {
    printf( "File read in. \n" );
    printf( "mag = %d xsize = %d ysize = %d blocksmaxize = %d\n",
            mag, xsize, ysize, maxblocksize );
    i = 0;
    for( this = top; this != 0; this=this->next) {
        printf( " T#%d  dx=%d dy=%d rx=%d ry=%d orient=%d a=%f b=%f\n",
                 i, this->dx, this->dy, this->rx, this->ry,
                 this->orient, this->a, this->b );
        i++;
    }
}


/* Set up picture array */

for( y = 0; y < ysize; y++ ) {
    for( x = 0; x < xsize; x++ ) {
        PICT(x, y) = 128;   /* Intermediate grey value */
    }
}
if( verbose ) printf( "Array inited.\n" );

/* Iterate and transform */

for( i = 0; i < iterations; i++ ) {

    for( this=top; this != 0; this=this->next ) {
        apply_transform( this );
    }
    temp = pictdata;
    pictdata = pictdata2;
    pictdata2 = temp;
    if( verbose ) printf( "Iteration done.\n" );
}

/* Output data--first free up core */
```

```
    free( pictdata2 );
    for( this=top->next; top != 0; top=this ) {
        this=top->next;
        free( top );
    }
    this=top=0;

    bmp = new_bmp( xsize, ysize );
    if( bmp == NULL ) {
        fprintf( stderr, "Out of Memory!\n" );
        exit( -1 );
    }

    datap = bmp->data;

    for( y = 0; y < ysize; y++ ) {
        for( x = 0; x < xsize; x++ ) {
            if( PICT(x, y) < 0 ) PICT(x, y) = 0;
            if( PICT(x, y) > 255) PICT(x, y)=255;
            *datap =  (unsigned char) PICT(x, y);
            datap++;
        }
        datap += (ysize % 4);   /* Correct any misalignment at each row */

    }

    /* Write out the file */

    outfilep = fopen( outfile, "w" );
    if( outfilep == NULL ) {
        fprintf(stderr, "Error opening output file; not written!\n" );
        exit( -1 );
    }

    if( bmp == NULL ) {
        fprintf(stderr, "bmp is NULL!!!!\n" );
    }
    write_bmp( outfilep, bmp );
    fclose( outfilep );
    if( verbose ) printf( "Done -- %s written.\n", outfile );
}

/* Apply a single transform to a single block; since this gets done a lot, */
/* the case statement is outside the loops rather than inside them. */

void apply_transform( struct transform *which ) {
    int rx;
    int ry;
    int dx;
    int dy;
    int blocksize;
    int dblocklen;
    int slow;
    int fast;
    float a;
    float b;

    rx = which->rx;
    ry = which->ry;
    dx = which->dx;
    dy = which->dy;
    blocksize = which->blocksize;
    dblocklen = 2*(blocksize-1);
    a = which->a;
    b = which->b;

    switch( which->orient ) {
    case 0:
```

```
    for( slow = 0; slow < blocksize; slow++ ) {
        for( fast = 0; fast < blocksize; fast++ ) {
            PICT2(rx+slow, ry+fast) = a * ( 2 +
                    PICT(dx+slow*2, dy+fast*2) +
                    PICT(dx+slow*2+1, dy+fast*2) +
                    PICT(dx+slow*2, dy+fast*2+1) +
                    PICT(dx+slow*2+1, dy+fast*2+1) ) / 4 +
                    b;
        }
    }
    break;
case 1:
    for( slow = 0; slow < blocksize; slow++ ) {
        for( fast = 0; fast < blocksize; fast++ ) {
            PICT2(rx+fast, ry+slow) = a * ( 2 +
                    PICT(dx+slow*2, dy+fast*2) +
                    PICT(dx+slow*2+1, dy+fast*2) +
                    PICT(dx+slow*2, dy+fast*2+1) +
                    PICT(dx+slow*2+1, dy+fast*2+1) ) / 4 +
                    b;
        }
    }
    break;
case 2:
    for( slow = 0; slow < blocksize; slow++ ) {
        for( fast = 0; fast < blocksize; fast++ ) {
            PICT2(rx+slow, ry+fast) = a * ( 2 +
                    PICT(dx+dblocklen-slow*2, dy+fast*2) +
                    PICT(dx+dblocklen-slow*2+1, dy+fast*2) +
                    PICT(dx+dblocklen-slow*2, dy+fast*2+1) +
                    PICT(dx+dblocklen-slow*2+1, dy+fast*2+1) ) / 4 +
                    b;
        }
    }
    break;
case 3:
    for( slow = 0; slow < blocksize; slow++ ) {
        for( fast = 0; fast < blocksize; fast++ ) {
            PICT2(rx+fast, ry+slow) = a * ( 2 +
                    PICT(dx+dblocklen-slow*2, dy+fast*2) +
                    PICT(dx+dblocklen-slow*2+1, dy+fast*2) +
                    PICT(dx+dblocklen-slow*2, dy+fast*2+1) +
                    PICT(dx+dblocklen-slow*2+1, dy+fast*2+1) ) / 4 +
                    b;
        }
    }
    break;
case 4:
    for( slow = 0; slow < blocksize; slow++ ) {
        for( fast = 0; fast < blocksize; fast++ ) {
            PICT2(rx+slow, ry+fast) = a * ( 2 +
                    PICT(dx+slow*2, dy+dblocklen-fast*2) +
                    PICT(dx+slow*2+1, dy+dblocklen-fast*2) +
                    PICT(dx+slow*2, dy+dblocklen-fast*2+1) +
                    PICT(dx+slow*2+1, dy+dblocklen-fast*2+1) ) / 4 +
                    b;
        }
    }
    break;
case 5:
    for( slow = 0; slow < blocksize; slow++ ) {
        for( fast = 0; fast < blocksize; fast++ ) {
            PICT2(rx+fast, ry+slow) = a * ( 2 +
                    PICT(dx+slow*2, dy+dblocklen-fast*2) +
                    PICT(dx+slow*2+1, dy+dblocklen-fast*2) +
                    PICT(dx+slow*2, dy+dblocklen-fast*2+1) +
                    PICT(dx+slow*2+1, dy+dblocklen-fast*2+1) ) / 4 +
                    b;
```

```
            }
        }
        break;
    case 6:
        for( slow = 0; slow < blocksize; slow++ ) {
            for( fast = 0; fast < blocksize; fast++ ) {
                PICT2(rx+slow, ry+fast) = a * ( 2 +
                        PICT(dx+dblocklen-slow*2, dy+dblocklen-fast*2) +
                        PICT(dx+dblocklen-slow*2+1, dy+dblocklen-fast*2) +
                        PICT(dx+dblocklen-slow*2, dy+dblocklen-fast*2+1) +
                        PICT(dx+dblocklen-slow*2+1, dy+dblocklen-fast*2+1) ) / 4 +
                        b;
            }
        }
        break;
    case 7:
        for( slow = 0; slow < blocksize; slow++ ) {
            for( fast = 0; fast < blocksize; fast++ ) {
                PICT2(rx+fast, ry+slow) = a * ( 2 +
                        PICT(dx+dblocklen-slow*2, dy+dblocklen-fast*2) +
                        PICT(dx+dblocklen-slow*2+1, dy+dblocklen-fast*2) +
                        PICT(dx+dblocklen-slow*2, dy+dblocklen-fast*2+1) +
                        PICT(dx+dblocklen-slow*2+1, dy+dblocklen-fast*2+1) ) / 4 +
                        b;
            }
        }
        break;
    default:
        fprintf( stderr, "Error--bad orient!\n" );
        break;
    }

    /* I have found that clamping the values every time through generally */
    /* leads to better convergence and output image quality, although it */
    /* is a bit slower. */

    for( slow = 0; slow < blocksize; slow++ ) {
        for( fast = 0; fast < blocksize; fast++ ) {
            if( PICT2( rx+slow, ry+fast ) < 0 ) {
                PICT2( rx+slow, ry+fast ) = 0;
            }
            if( PICT2( rx+slow, ry+fast ) > 255 ) {
                PICT2( rx+slow, ry+fast ) = 255;
            }
        }
    }
}

/* Read a file bit by bit.  This is used to read the quadtree map. */

int read_bit() {
    static unsigned char this_char = 0;
    static int bits_left = 0;

    if( bits_left <= 0 ) {
        read( infilenum, &this_char, 1 );
        bits_left = 8;
    }

    bits_left--;
    if( (this_char & 0x01) == 0 ) {
        this_char >>= 1;
        return ZERO_BIT;
    } else {
        this_char >>= 1;
        return ONE_BIT;
    }
}
```

```
/* This is a recursive function, "unrolling" the tree and creating the */
/* transform structures.  The block size is filled in at this point. */

void read_qt_map( int x, int y, int blksize ) {
    if( blksize <= 3 ) {
        /* In this case, we need to simply create the transform structure */
        last->next =(struct transform *) malloc( sizeof( struct transform) );
        last=last->next;
        if( last == 0 ) {
            fprintf( stderr, "Error: out of memory!\n" );
            exit( 1 );
        }
        last->rx = x;
        last->ry = y;
        last->blocksize = blksize;

    } else {
        /* Read the map and do what it says. */
        if( read_bit() == ZERO_BIT ) {
            /* Create a transform structure */
            last->next =(struct transform *) malloc( sizeof( struct transform) );
            last=last->next;
            if( last == 0 ) {
                fprintf( stderr, "Error: out of memory!\n" );
                exit( 1 );
            }
            last->rx = x;
            last->ry = y;
            last->blocksize = blksize;
        } else {
            fflush( stdout );
            /* Recursively create transforms */
            blksize /= 2;
            read_qt_map( x, y, blksize );
            read_qt_map( x+blksize, y, blksize );
            read_qt_map( x, y+blksize, blksize );
            read_qt_map( x+blksize, y+blksize, blksize );
        }
    }
}

/* Fill in a transform structure from data in the file. */

void read_transform( struct transform *this ) {
    struct filebit bit;

    read( infilenum, &bit, sizeof( struct filebit ) );

    this->dx = bit.dx;
    this->dy = bit.dy;
    this->a = bit.a;
    this->b = bit.b;
    this->orient = bit.orient;
}
```

A.4 Bitmap File Library

A.4.1 Header File: bmp.h

```
/* Stuff for parsing .BMP files */

/* Approximate definitions.  We'll have to do junk to fix up x86 endianness. */
/* THESE ARE PLATFORM DEPENDANT:  THESE WILL WORK FOR MANY 32 BIT SYSTEMS. */
```

```
typedef unsigned short UINT;
typedef signed short WORD;
typedef unsigned int DWORD;
typedef unsigned char BYTE;
typedef signed int LONG;

/* constants for the biCompression field */
#define BI_RGB      0
#define BI_RLE8     1
#define BI_RLE4     2

#define CORRECT_BF_TYPE 19778

/* A really useful macro */
#define ROWLEN( bmpp ) ((3+(bmpp)->bmi->bmih.biWidth)&(~0x03))

/* RGB values */
typedef struct a_RGBQUAD {     /* rgbq */
    BYTE    Blue;
    BYTE    Green;
    BYTE    Red;
    BYTE    Reserved;
} RGBQUAD;

/* A Bitmap file is organized as following:
        BITMAPFILEHEADER
        BITMAPINFO
                BITMAPINFOHEADER
                [RGBQUAD] if required
    actual data */

struct bmpfile {
    struct bmfh *bmfh;
    struct bmi *bmi;
    unsigned char *data;
};

struct bmfh {    /* BITMAPFILEHEADER */
    UINT    bfType;     /* Must be "BM" */
    DWORD   bfSize;     /* Size of file, bytes */
    UINT    bfReserved1;    /* Zero */
    UINT    bfReserved2;    /* Zero */
    DWORD   bfOffBits;      /* Offset of actual picture data */
} ;

struct bmih {    /* BITMAPINFOHEADER */
    DWORD   biSize;     /* Sizeof (struct bmih ) */
    LONG    biWidth;        /* Width in pixels of bitmap */
    LONG    biHeight;       /* Height in pixels of bitmap */
    WORD    biPlanes;       /* Pixel planes--must be 1 */
    WORD    biBitCount;     /* 1, 4, 8, 24 -- we use 8 */
    DWORD   biCompression;  /* BI_RLE8, BI_RLE4, BI_RGB */
    DWORD   biSizeImage;    /* Size of actual image */
    LONG    biXPelsPerMeter;    /* resolution in X direction */
    LONG    biYPelsPerMeter;    /* resolution in Y direction */
    DWORD   biClrUsed;      /* Number of used colors */
    DWORD   biClrImportant; /* Number of important colors */
} ;

struct bmi {  /* BITMAPINFO */
    struct bmih    bmih;
    RGBQUAD        bmiColors[256];  /* Yecch--this should be dynamic. */
} ;

/* Public functions: */

/* Read in stuff from a file; does not handle RLE, but does everything else. */
```

```
struct bmpfile *read_bmp( FILE *f );
/* Write stuff out to a file */
int write_bmp( FILE *f, struct bmpfile *bmp );
/* Create a new bitmap with "random" data */
struct bmpfile *new_bmp( int xsize, int ysize );
/* Get rid of a bitmap structure */
void dispose_bmp( struct bmpfile *bmp );
/* Format it for use with our program--make it properly greyscale. */
int cleanup_bmp( struct bmpfile *bmp );
/* Print in a human-readable (?) form */
void dump_bmp( struct bmpfile *bmp );
```

A.4.2 Source File: bmp.c

```
#include <stdlib.h>
#include <stdio.h>
#include "bmp.h"

/* Utility function prototypes */
static UINT read_UINT( FILE *f );
static WORD read_WORD( FILE *f );
static DWORD read_DWORD( FILE *f );
static BYTE read_BYTE( FILE *f );
static LONG read_LONG( FILE *f );

static void write_UINT( FILE *f, UINT what );
static void write_WORD( FILE *f, WORD what );
static void write_DWORD( FILE *f, DWORD what );
static void write_BYTE( FILE *f, BYTE what );
static void write_LONG( FILE *f, LONG what );
void dump_bmp( struct bmpfile *it );

/* Reading various Wintel base types; these should be platform independent. */

static BYTE read_BYTE( FILE *f ) {
    return getc( f ) ;
}

static void write_BYTE( FILE *f, BYTE what ) {
    putc( what, f );
}

static WORD read_WORD( FILE *f ) {
    int a,b;

    a = getc( f );
    b = getc( f );
/*  printf( "read_WORD:  bytes %02X %02X,        word = %04X\n",
            (int) a, (int) b, (b * 256) + a ); */
    return (b * 256) + a;
}

static void write_WORD( FILE *f, WORD what ) {
/*  printf( "write_WORD:  bytes %02X %02X,        word = %04X\n",
            (int) what%256, (int) what/256, what ); */
    putc( what % 256, f );
    putc( what / 256, f );
}

static DWORD read_DWORD( FILE *f ) {
    int a,b,c,d;
    DWORD dw;

    a = getc( f );  b = getc( f );
    c = getc( f );  d = getc( f );
```

```
/*  printf( "read_DWORD: bytes %02X %02X %02X %02X, dword = %08X\n",
            (int) a, (int) b, (int) c, (int) d, (d * 256 + c) * 65536 + b * 256 + a ); */
    return (d * 256 + c) * 65536 + b * 256 + a;
}

static void write_DWORD( FILE *f, DWORD what ) {
    int a, b, c, d;

    a = what % 256;
    b = (what / 256) % 256;
    what /= 65536;
    c = what % 256;
    d = (what / 256 );
/*  printf( "write_DWORD: bytes %02X %02X %02X %02X,  long = %08X\n",
            (int) a, (int) b, (int) c, (int) d, (d * 256 + c) * 65536 + b * 256 + a );  */

    putc( a, f );   putc( b, f );
    putc( c, f );   putc( d, f );
}

static LONG read_LONG( FILE *f ) {
    int a,b,c,d;

    a = getc( f );  b = getc( f );
    c = getc( f );  d = getc( f );

/*  printf( "read_LONG:  bytes %02X %02X %02X %02X,  long = %08X\n",
            (int) a, (int) b, (int) c, (int) d, (d * 256 + c) * 65536 + b * 256 + a ); */
    return (d * 256 + c) * 65536 + b * 256 + a;
}

static void write_LONG( FILE *f, LONG what ) {
    int a, b, c, d;

    a = what % 256;
    b = (what / 256) % 256;
    what /= 65536;
    c = what % 256;
    d = (what / 256 );
/*  printf( "write_LONG:  bytes %02X %02X %02X %02X,  long = %08X\n",
            (int) a, (int) b, (int) c, (int) d, (d * 256 + c) * 65536 + b * 256 + a );  */

    putc( a, f );   putc( b, f );
    putc( c, f );   putc( d, f );
}

static UINT read_UINT( FILE *f ) {
    int a,b;

    a = getc( f );
    b = getc( f );
/*  printf( "read_UINT:  bytes %02X %02X,        uint = %04X\n",
            (int) a, (int) b, (b * 256) + a ); */
    return (b * 256) + a;
}

static void write_UINT( FILE *f, UINT what ) {
/*  printf( "write_UINT:  bytes %02X %02X,        word = %04X\n",
            (int) what%256, (int) what/256, what ); */
    putc(what % 256, f );
    putc(what / 256, f );
}

/****************** Public Functions **********************/
/* Get rid of a bitmap */
void dispose_bmp( struct bmpfile *bmp ) {
    if( bmp != NULL ) {
```

```
        if( bmp->bmfh != NULL ) {
            free( bmp->bmfh ) ;
        }
        if( bmp->bmi != NULL ) {
            free( bmp->bmi );
        }
        if( bmp->data != NULL ) {
            free( bmp->data );
        }
        free( bmp );
    }
}

/* Create a new bitmap in memory */
struct bmpfile *new_bmp( int xsize, int ysize ) {
    struct bmpfile *it;
    int i;

    it = (struct bmpfile *)malloc( sizeof( struct bmpfile ) );
    if( it != NULL ) {
        it->bmfh = (struct bmfh *)malloc( sizeof( struct bmfh ) );
        it->bmi = (struct bmi *)malloc( sizeof( struct bmi ) );
        /* Data must be aligned mod four */
        it->data = (unsigned char *)malloc( xsize * (4*(ysize + 3 ) / 4 ) );
        if( it->bmfh == NULL || it->bmi == NULL || it->data == NULL ) {
            dispose_bmp( it );
            it = NULL;
        }
    }
    if( it == NULL ) return it; /* Not enough core */

    /* Fill in bitmap header */
    it->bmfh->bfType = CORRECT_BF_TYPE;
    it->bmfh->bfReserved1 = 0;
    it->bmfh->bfReserved2 = 0;
    it->bmfh->bfOffBits = sizeof( struct bmfh) + sizeof( struct bmi );
    it->bmfh->bfSize = it->bmfh->bfOffBits + xsize * (4*(ysize + 3) / 4 );

    /* Fill in bitmap info header */
    it->bmi->bmih.biSize = sizeof( struct bmih );   /* This should work */
    it->bmi->bmih.biWidth = xsize;
    it->bmi->bmih.biHeight = ysize;
    it->bmi->bmih.biPlanes = 1;
    it->bmi->bmih.biBitCount = 8;
    it->bmi->bmih.biCompression = BI_RGB;
    it->bmi->bmih.biSizeImage = xsize * (4*(ysize+3)/4);
    it->bmi->bmih.biXPelsPerMeter = 2800;   /* somewhere near 72 dpi */
    it->bmi->bmih.biYPelsPerMeter = 2800;
    it->bmi->bmih.biClrUsed = 256;  /* all of them */
    it->bmi->bmih.biClrImportant = 256; /* all of them */

    /* Fill in the color quads */
    for( i = 0; i < 256; i++ ) {
        it->bmi->bmiColors[i].Red = i;
        it->bmi->bmiColors[i].Green = i;
        it->bmi->bmiColors[i].Blue = i;
        it->bmi->bmiColors[i].Reserved = 0;
    }

    /* We will not touch the initial picture data; use random data */
    return it;
}

/* Write a bitmap to a file */
int write_bmp (FILE *outfile, struct bmpfile *it ) {
    long i;
    int ColorEntries;
    unsigned char *datap;
```

```
/* dump_bmp( it ); */

if( outfile == NULL ) fprintf( stderr, "ERROR: NULL OUTFILE!\n" );

/* Write out the header */
write_UINT( outfile, it->bmfh->bfType );
write_DWORD( outfile, it->bmfh->bfSize );
write_UINT( outfile, it->bmfh->bfReserved1 );
write_UINT( outfile, it->bmfh->bfReserved2 );
write_DWORD( outfile, it->bmfh->bfOffBits );

/* Verify the number of color entries */
ColorEntries = it->bmi->bmih.biClrUsed;
if( ColorEntries > 256 ) {
    fprintf(stderr, "Warning:  more than 256 color indices; assuming maximum.\n" );
    ColorEntries = 0;
    it->bmi->bmih.biClrUsed = 0;
}

/* Write out the bitmap info header */
write_DWORD( outfile, it->bmi->bmih.biSize );
write_LONG( outfile, it->bmi->bmih.biWidth );
write_LONG( outfile, it->bmi->bmih.biHeight );
write_WORD( outfile, it->bmi->bmih.biPlanes );
write_WORD( outfile, it->bmi->bmih.biBitCount );
write_DWORD( outfile, it->bmi->bmih.biCompression );
write_DWORD( outfile, it->bmi->bmih.biSizeImage );
write_LONG( outfile, it->bmi->bmih.biXPelsPerMeter );
write_LONG( outfile, it->bmi->bmih.biYPelsPerMeter );
write_DWORD( outfile, it->bmi->bmih.biClrUsed );
write_DWORD( outfile, it->bmi->bmih.biClrImportant );

/* Write out the color quads */
if( ColorEntries == 0 ) {
    switch( it->bmi->bmih.biBitCount ) {
    case 1:
        ColorEntries = 2;
        break;
    case 4:
        ColorEntries = 16;
        break;
    case 8:
        ColorEntries = 256;
        break;
    case 24:
        ColorEntries = 0;
        break;
    default:
        fprintf(stderr, "Warning:  Bad bit depth in .bmp file\n" );
        ColorEntries = 0;
    }
}


for( i = 0; i < ColorEntries; i++ ) {
    write_BYTE( outfile, it->bmi->bmiColors[i].Blue );
    write_BYTE( outfile, it->bmi->bmiColors[i].Green );
    write_BYTE( outfile, it->bmi->bmiColors[i].Red );
    write_BYTE( outfile, it->bmi->bmiColors[i].Reserved );
}

/* Write out actual image data */

datap = it->data;
for( i = 0; i < it->bmi->bmih.biSizeImage; i++ ) {
    write_BYTE( outfile, *datap++ );
}
```

```
    return 0;
}


/* Read a bitmap from a file */
struct bmpfile *read_bmp( FILE *infile ) {
    struct bmpfile *it;
    long i;
    int ColorEntries;
    unsigned char *datap;

    it = (struct bmpfile *)malloc( sizeof( struct bmpfile ) );
    if( it != NULL ) {
        it->bmfh = (struct bmfh *)malloc( sizeof( struct bmfh ) );
        it->bmi = (struct bmi *)malloc( sizeof( struct bmi ) );
        /* Data must be aligned mod four */
        it->data = NULL;
        if( it->bmfh == NULL || it->bmi == NULL ) {
            dispose_bmp( it );
            it = NULL;
        }
    }

    if( it == NULL ) return it; /* Not enough core */

    /* Fill in bitmap header */
    it->bmfh->bfType = read_UINT( infile );
    it->bmfh->bfSize = read_DWORD( infile );
    it->bmfh->bfReserved1 = read_UINT( infile );
    it->bmfh->bfReserved2 = read_UINT( infile );
    it->bmfh->bfOffBits = read_DWORD( infile );

    /* Check some stuff */
    if( it->bmfh->bfType != CORRECT_BF_TYPE ) {
        fprintf( stderr, "Warning:  Incorrect magic number for bmp file. \n" );
        /* We probably ought to quit right about now, but... */
    }

    /* Fill in bitmap info header */
    it->bmi->bmih.biSize = read_DWORD( infile );
    it->bmi->bmih.biWidth = read_LONG( infile );
    it->bmi->bmih.biHeight = read_LONG( infile );
    it->bmi->bmih.biPlanes = read_WORD( infile );
    it->bmi->bmih.biBitCount = read_WORD( infile );
    it->bmi->bmih.biCompression = read_DWORD( infile );
    it->bmi->bmih.biSizeImage = read_DWORD( infile );
    it->bmi->bmih.biXPelsPerMeter = read_LONG( infile );
    it->bmi->bmih.biYPelsPerMeter = read_LONG( infile );
    it->bmi->bmih.biClrUsed = read_DWORD( infile );
    it->bmi->bmih.biClrImportant = read_DWORD( infile );

    /* Verify a couple of things */
    if( it->bmi->bmih.biSize != sizeof( struct bmih ) ) {
        fprintf( stderr, "Warning:  Incorrect bitmap info header size.\n" );
    }
    if( it->bmi->bmih.biClrImportant > it->bmi->bmih.biClrUsed ) {
        fprintf( stderr, "Warning:  More important colors than used colors.\n" );
    }

    switch( it->bmi->bmih.biBitCount ) {
    case 1:
        if( it->bmi->bmih.biClrUsed > 2 ) {
            fprintf( stderr, "Warning:  More than two colors for 1-bit image.\n" );
        }
        if( it->bmi->bmih.biCompression != BI_RGB ) {
            fprintf( stderr, "Warning:  Unknown compression for 1-bit image.\n" );
        }
```

```
    break;
case 4:
    if( it->bmi->bmih.biClrUsed > 16 ) {
        fprintf( stderr, "Warning:  More than sixteen colors for 4-bit image.\n" );
    }
    if( it->bmi->bmih.biCompression != BI_RGB && it->bmi->bmih.biCompression != BI_RLE4 ) {
        fprintf( stderr, "Warning:  Unknown compression for 4-bit image.\n" );
    }
    break;
case 8:
    if( it->bmi->bmih.biCompression != BI_RGB && it->bmi->bmih.biCompression != BI_RLE8 ) {
        fprintf( stderr, "Warning:  Unknown compression for 8-bit image.\n" );
    }
    break;
case 24:
    if( it->bmi->bmih.biCompression != BI_RGB ) {
        fprintf( stderr, "Warning:  Unknown compresion for true color image.\n" );
    }
    break;
default:
    fprintf( stderr, "Warning:  Incorrect color depth.\n" );

}

/* Fill in the color quads */
ColorEntries = it->bmi->bmih.biClrUsed;
if( ColorEntries > 256 ) {
    fprintf(stderr, "Warning:  more than 256 color indices; assuming maximum.\n" );
    ColorEntries = 0;
    it->bmi->bmih.biClrUsed = 0;
}
if( ColorEntries == 0 ) {
    switch( it->bmi->bmih.biBitCount ) {
    case 1:
        ColorEntries = 2;
        break;
    case 4:
        ColorEntries = 16;
        break;
    case 8:
        ColorEntries = 256;
        break;
    case 24:
        ColorEntries = 0;
        break;
    default:
        fprintf(stderr, "Warning:  Bad bit depth in .bmp file\n" );
        ColorEntries = 0;
    }
}

for( i = 0; i < ColorEntries; i++ ) {
    it->bmi->bmiColors[i].Blue = read_BYTE( infile );
    it->bmi->bmiColors[i].Green = read_BYTE( infile );
    it->bmi->bmiColors[i].Red = read_BYTE( infile );
    it->bmi->bmiColors[i].Reserved = read_BYTE( infile );
}

/* Read in actual image data */
if( it->bmi->bmih.biSizeImage == 0 ) {
    it->bmi->bmih.biSizeImage = it->bmfh->bfSize - it->bmfh->bfOffBits;
}

it->data = (unsigned char *)malloc( it->bmi->bmih.biSizeImage );
if( it->data == NULL ) {
    dispose_bmp( it );
    it = NULL;
    return it;
```

```
    }

    datap = it->data;

    for( i = 0; i < it->bmi->bmih.biSizeImage; i++ ) {
        *datap++ = read_BYTE( infile );
    }

    return it;
}

int cleanup_bmp( struct bmpfile *bmp ) {
    long i;
    unsigned char conv[256];
    unsigned char *datap;

    /* Check a couple of essentials */
    if( bmp->bmi->bmih.biBitCount != 8 ) {
        fprintf( stderr, "Error--cleanup() only works with 8-bit bitmaps.\n" );
        return -1;
    }
    if( bmp->bmi->bmih.biCompression != BI_RGB ) {
        fprintf( stderr, "Error--cleanup() doesn't grok RLE at the momment.\n" );
        return -2;
    }

    /* Read in translation array */
    for( i = 0; i < bmp->bmi->bmih.biClrUsed; i++ ) {
        if( bmp->bmi->bmiColors[i].Red != bmp->bmi->bmiColors[i].Green ||
                bmp->bmi->bmiColors[i].Red != bmp->bmi->bmiColors[i].Blue )  {
            fprintf( stderr, "Warning: Color #%d is not a greyscale value.\n", i );
        }
        conv[i] = bmp->bmi->bmiColors[i].Red;
    }

    /* Now make the "correct" translation array. */
    for( i = 0; i < 256; i++ ) {
        bmp->bmi->bmiColors[i].Red = i;
        bmp->bmi->bmiColors[i].Green = i;
        bmp->bmi->bmiColors[i].Blue = i;
        bmp->bmi->bmiColors[i].Reserved = 0;
    }

    bmp->bmi->bmih.biClrUsed = 256;

    /* Now, fix our actual image data */

    datap = bmp->data;
    for( i = 0; i < bmp->bmi->bmih.biSizeImage; i++ ) {
        *datap = conv[*datap];
        datap++;
    }

    return 0;
}

void dump_bmp( struct bmpfile *bitmap ) {
    int i;

    printf( "Bitmap File Header:\n" );
    printf( "\tbfType      = %d\n", bitmap->bmfh->bfType );
    printf( "\tbfSize      = %d\n", bitmap->bmfh->bfSize );
    printf( "\tbfReserved1 = %d\n", bitmap->bmfh->bfReserved1 );
    printf( "\tbfReserved2 = %d\n", bitmap->bmfh->bfReserved2 );
    printf( "\tbfOffBits   = %d\n", bitmap->bmfh->bfOffBits );

    printf( "\nBitmap Info Header:\n" );
```

```
            printf( "\tbiSize          = %d\n", bitmap->bmi->bmih.biSize );
            printf( "\tbiWidth         = %d\n", bitmap->bmi->bmih.biWidth );
            printf( "\tbiHeight        = %d\n", bitmap->bmi->bmih.biHeight );
            printf( "\tbiPlanes        = %d\n", bitmap->bmi->bmih.biPlanes );
            printf( "\tbiBitcount      = %d\n", bitmap->bmi->bmih.biBitCount );
            printf( "\tbiCompression   = %d\n", bitmap->bmi->bmih.biCompression );
            printf( "\tbiSizeImage     = %d\n", bitmap->bmi->bmih.biSizeImage );
            printf( "\tbiXPelsPerMeter = %d\n", bitmap->bmi->bmih.biXPelsPerMeter );
            printf( "\tbiYPelsPerMeter = %d\n", bitmap->bmi->bmih.biYPelsPerMeter );
            printf( "\tbiClrUsed       = %d\n", bitmap->bmi->bmih.biClrUsed );
            printf( "\tbiClrImportant  = %d\n", bitmap->bmi->bmih.biClrImportant );

            printf( "\nColors:\n" );
            for( i = 0; i < 64; i++ ) {
                printf( "#%3u: 0x%02X%02X%02X/%02X   #%3u: 0x%02X%02X%02X/%02x "
                        "  #%3u: 0x%02X%02X%02X/%02X   #%3u: 0x%02X%02X%02X/%02x\n",
                        i, (int) bitmap->bmi->bmiColors[i].Red,
                           (int) bitmap->bmi->bmiColors[i].Green,
                           (int) bitmap->bmi->bmiColors[i].Blue,
                           (int) bitmap->bmi->bmiColors[i].Reserved,
                        i+64, (int) bitmap->bmi->bmiColors[i+64].Red,
                           (int) bitmap->bmi->bmiColors[i+64].Green,
                           (int) bitmap->bmi->bmiColors[i+64].Blue,
                           (int) bitmap->bmi->bmiColors[i+64].Reserved,
                        i+128, (int) bitmap->bmi->bmiColors[i+128].Red,
                           (int) bitmap->bmi->bmiColors[i+128].Green,
                           (int) bitmap->bmi->bmiColors[i+128].Blue,
                           (int) bitmap->bmi->bmiColors[i+128].Reserved,
                        i+192, (int) bitmap->bmi->bmiColors[i+192].Red,
                           (int) bitmap->bmi->bmiColors[i+192].Green,
                           (int) bitmap->bmi->bmiColors[i+192].Blue,
                           (int) bitmap->bmi->bmiColors[i+192].Reserved );

            }

}       }
```

Appendix B

Synthesis Script File

The scripts are separated into a few sections, to allow easier compilation of various portions of the system. The Master script file calls these various sections in order, producing a final design.

B.1 Master Script File: syn.scr2

```
include syn.scr2a /* Compile the memory interface unit */
remove_design /* Remove the stuff from memory */
include syn.scr2b /* Compile the parameter computation block */
remove_design /* Remove the stuff from memory */
include syn.scr2c /* Compile the rest of the pipeline unit */
remove_design /* Remove the stuff from memory */
include syn.scr2d /* Compile the host interface and top level */

/* Do the various reports */
report_timing >timing_report.txt
report_area > area_report.txt
report_resources > resource_report.txt
```

B.2 Memory Interface Unit: syn.scr2a

```
define_design_lib work -path syn_work2
read -format vhdl thesis_pkg.vhd

read -format vhdl den_comp_chunk.vhd
check_design
set_dont_use { dw01.sldb/DW01_sub/rpcs dw01.sldb/DW01_add/rpcs dw01.sldb/DW01_addsub/rpcs }

read -format vhdl block_addr_chunk.vhd
check_design

read -format vhdl memory_cache_chunk.vhd
check_design

read -format vhdl mem_iface_unit.vhd
check_design
create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
```

```
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high

ungroup -all -flatten
create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high
write -format db
```

B.3 Parameter Computation Chunk: syn.scr2b

```
define_design_lib work -path syn_work2
read -format vhdl thesis_pkg.vhd

read -format vhdl param_comp_chunk.vhd
check_design
set_dont_use { dw01.sldb/DW01_sub/rpcs dw01.sldb/OW01_add/rpcs dw01.sldb/OW01_addsub/rpcs }

create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high

ungroup -all -flatten
create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high

write -format db
```

B.4 Pipeline Unit: syn.scr2c

```
define_design_lib work -path syn_work2
read -format vhdl thesis_pkg.vhd

read -format vhdl MAC_chunk.vhd
check_design
set_dont_use { dw01.sldb/DW01_sub/rpcs dw01.sldb/DW01_add/rpcs dw01.sldb/DW01_addsub/rpcs }

create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
```

```
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high

ungroup -all -flatten
create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high
write -format db
remove_design MAC_CHUNK

read -format vhdl range_memory_syn.vhd
create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high

ungroup -all -flatten
create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high
write -format db
remove_design RANGE_MEMORY

read -format vhdl range_block_chunk.vhd
check_design
create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high
write -format db

read -format db MAC_CHUNK.db
read -format db RANGE_MEMORY.db
read -format db PARAM_COMP_CHUNK.db
read -format vhdl pipeline_unit.vhd
write -format db
```

B.5 Host Interface and Top Level: syn.scr2d

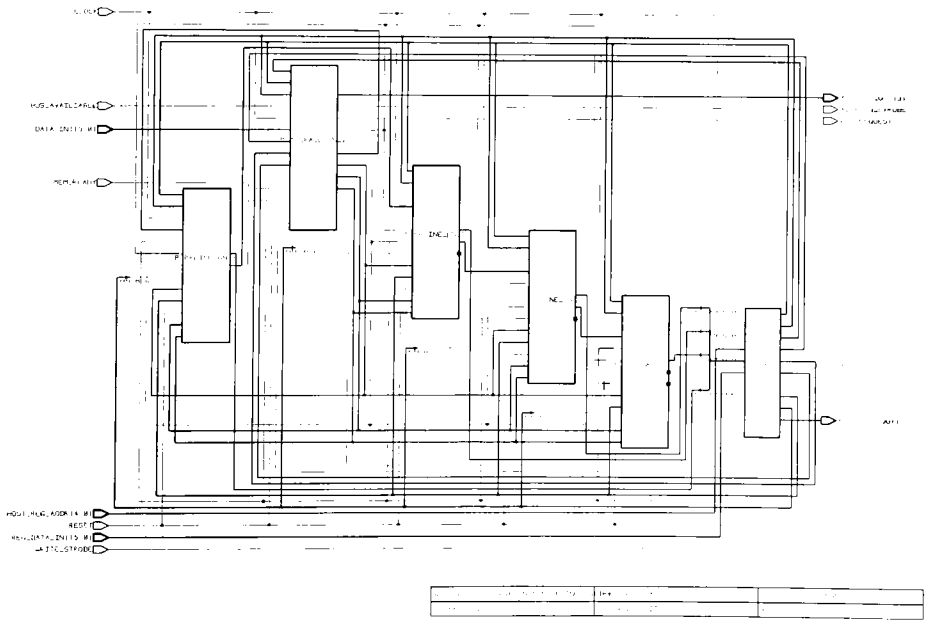
```
define_design_lib work -path syn_work2
read -format vhdl thesis_pkg.vhd
hdlin_auto_save_templates = TRUE
read -format vhdl host_iface_unit.vhd
read -format vhdl top_level.vhd

read -format db PIPELINE_UNIT.db
read -format db MEM_IFACE_UNIT.db
set_dont_touch PIPELINE_UNIT
set_dont_touch MEM_IFACE_UNIT

read -format vhdl syn_top_level.vhd
check_design
set_dont_use { dw01.sldb/DW01_sub/rpcs dw01.sldb/DW01_add/rpcs dw01.sldb/DW01_addsub/rpcs }

create_clock -period 50 CLOCK
set_drive 0 { CLOCK RESET }
set_dont_touch_network { CLOCK RESET }
set_attribute all_inputs() is_clk false -type boolean -quiet
set_attribute { CLOCK RESET} is_clk true -type boolean -quiet
set_input_delay 1 -clock CLOCK filter(all_inputs() "@is_clk != true")
set_output_delay 5 -clock CLOCK all_outputs()
set_load 5 all_outputs()
compile -map_effort high

write -format db
write -format db HOST_IFACE_UNIT_NUM_PIPELINE_UNITS4
write -format db TOP_LEVEL_NUM_PIPELINE_UNITS4
```

Appendix C

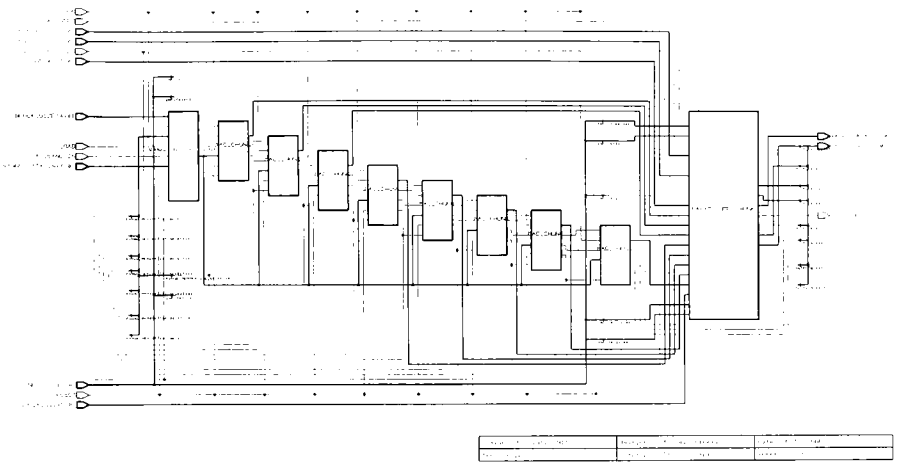
Schematics Generated by Synthesis

Some representative schematic diagrams, generated by Synopsys, are included below. These are only small portions of the design; the entire set of schematic diagrams would require several hundred pages.

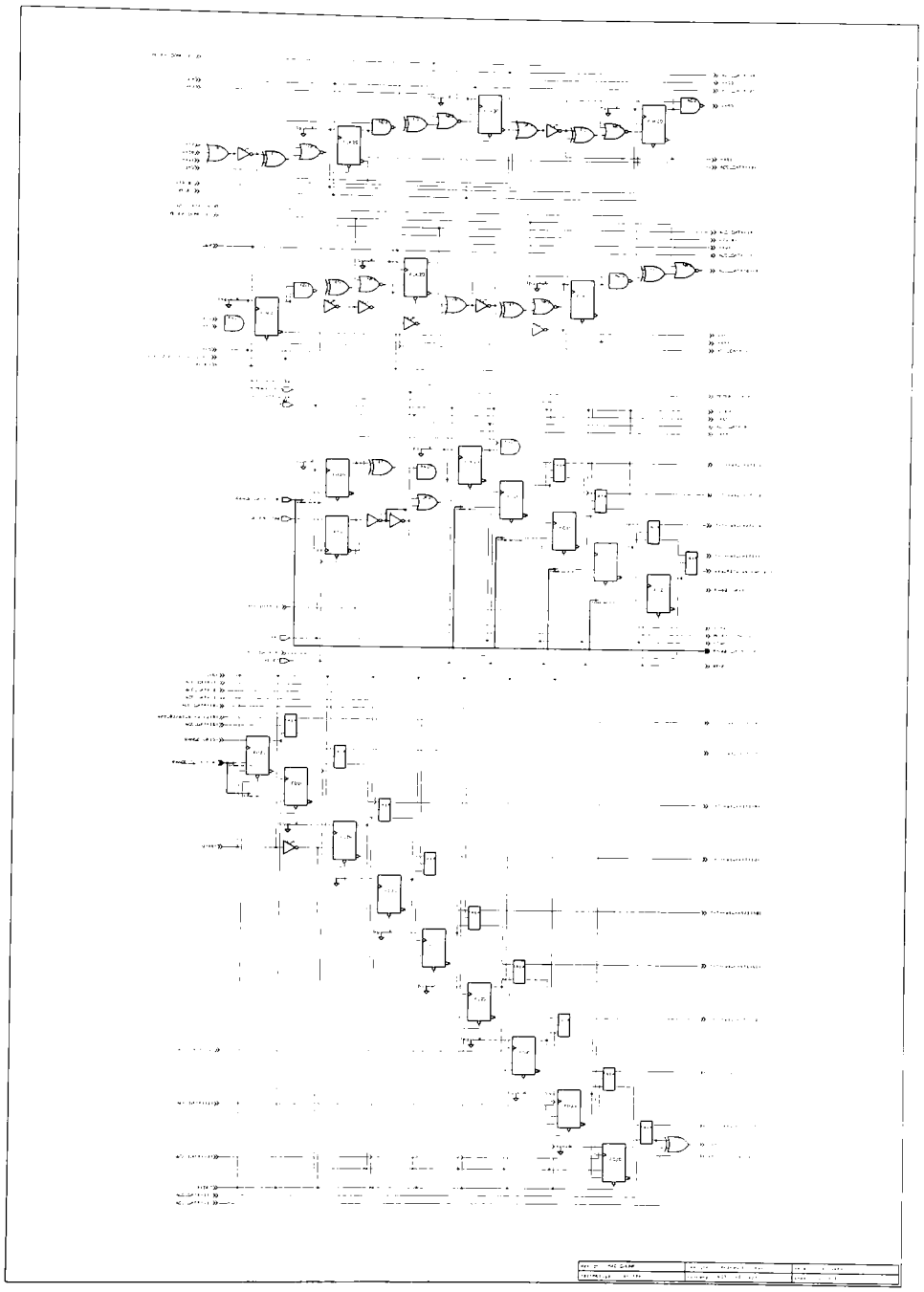
C.1 Top Level Schematic

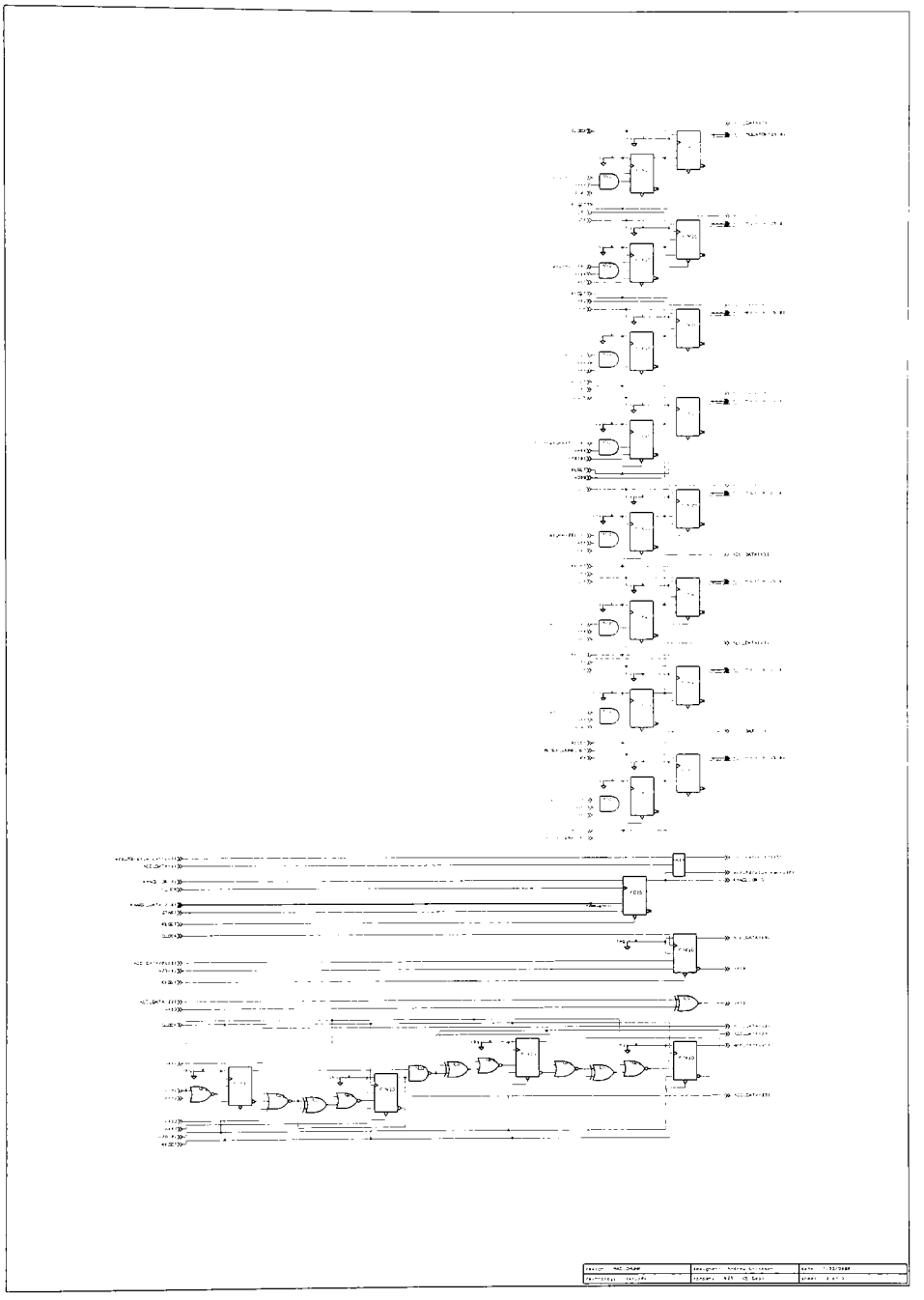


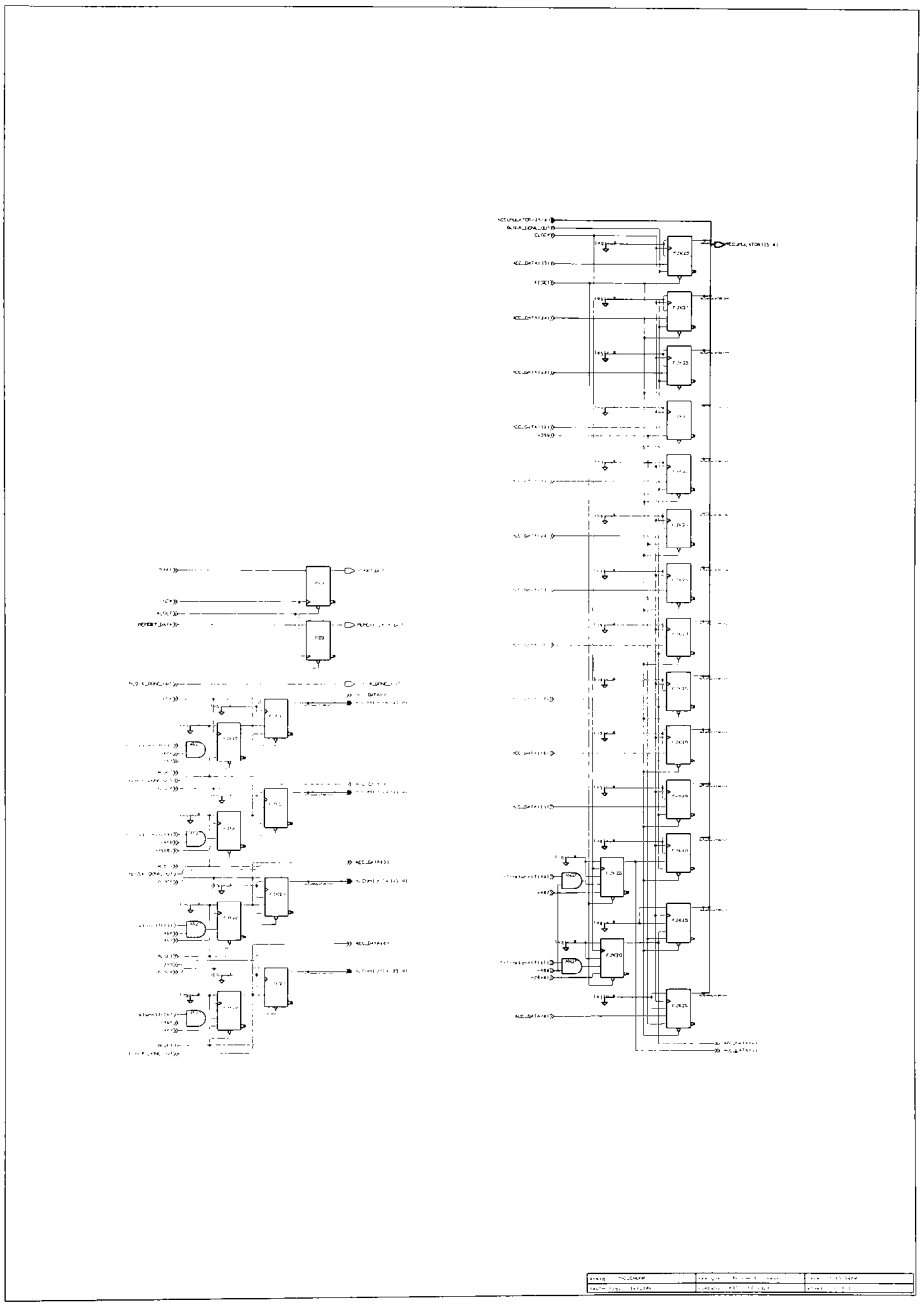
C.2 Pipeline Unit



C.3 MAC chunk







Appendix D

Image Manipulation Tools

Several small image manipulation tools were developed during the writing of this thesis; code for them is included on the included CD-ROM. This appendix gives a brief introduction to each of them and information on their operation.

D.1 bmp2memfile

This program converts a bitmap file (Windows .BMP format) into the form used read by the simulated main memory VHDL entity. This format is just a series of four-digit (two byte) hex numbers, each on its own line, without any other characters.

The program takes exactly two arguments: a bitmap file and an output file, in that order. The simulated memory expects the output from this program to be named `memory_data`. The program writes five words (ten bytes) of nulls before the image data; thus, during simulation, the image base address is 10 (decimal).

D.2 bmp2pgm

This program converts a grayscale bitmap file to the pgm format, which is commonly used as a (simple) intermediate format for translation into any of several image file formats. Image data in a pgm file is stored as the digits of decimal numbers (stored as text); thus, a pgm file takes up considerably more disk space than a bitmap file.

The program can take zero, one, or two command line arguments. If zero are provided, it acts as a filter, translating the standard input to the standard output. If a single argument is given, it is taken to be the input file (instead of standard input) and the output is placed on standard output. If two arguments are supplied, they are the input and output files, respectively.

D.3 bmpstat

This program provides a dump of the headers for a bitmap file—essentially, all the information contained in it except for the actual image data. This includes the size, the format of the data, and the color table.

One or more bitmap files must be supplied via the command line; information is printed for each.

D.4 imgdiff

This program compares two images and produces a new image where the grayscale value of each pixel is the absolute value of the difference between the corresponding pixels of the two original images. It thus provides a visual indication of errors between the two images.

Three command line arguments must be supplied: two input files and the output file name, in that order. The output file will have the same dimensions as the two input files, which must be of equal size. All image files are bitmap files.

D.5 imgcompare

This program provides a rudimentary histogram of differences of grayscale values between two equally sized images. Both of these images may be supplied, or one may be an (automatically generated) image consisting of all black; in that case, the histogram represents the relative distribution of the grayscale values of the supplied image. This program produces a text output (a crude histogram) which is designed to fit on a 132x24 terminal.

Two versions of the histogram are possible: a normal and a "scrunched" version, which differ in the size of the histogram bins. In the normal version, each bin holds a single difference value and only the first 130 are displayed. In the scrunched mode, each bin holds two difference values, allowing the display of all 256 possible difference values (at a loss of resolution).

Scrunched mode is selected by a `-s` flag as the first argument to the program; its absence indicates normal mode. One or two additional arguments specify the images to be compared; if only one is present, the all black image is used as the second and a histogram of grayscale values is produced.

Typically, scrunched mode is used when a single image is specified and normal mode is used when two (presumably similar) images are compared.

D.6 ftrans3

This program translates a text file indicating the transformations found by (simulated) hardware into my fractal image compression file format for later decompression by fdeocmp3. The input file format is designed for easy entry by hand

from a simulation; its lines consist solely of an increasing integer (primarily intended for detecting missing or duplicate lines), a space, the address of the best domain block found (assuming an offset of ten to the start of the image), a forward slash character, and the index of the isometry for the best transformation. No additional characters should be present. Successive lines are assumed to be successive range blocks in the image, working from low addresses to high addresses.

The program requires four arguments: the original bitmap file, the block size used, the results filename (whose contents are described above), and an output filename, in that order. When it runs, the program produces, on standard output, a list of the transformations and the MSE values associated with them.

D.7 fstat3

This program dumps some statistics on a fractally compressed image (stored in the file format of fcomp3 or ftrans3). Additionally, if the `-v` flag is used, the transforms are dumped. Any number of files may be specified on the command line.

The information dumped includes the image dimensions, the number of blocks of various sizes, and the maximum, minimum, and average values for $|a|$ and b.

Bibliography

[1] Barnsley, Michael F.; Hurd, Lyman P., *Fractal Image Compression*, A. K. Peters, Ltd., Wellesley, MA, 1993.

[2] Rolewicz, Stefan, *Metric Linear Spaces*, Panswowe Wydawnietwo Naukowe (PWN—Polish Scientific Publishers), Warszawa, Poland, 1972.

[3] Davis, G. M., "A Wavelet-Based Analysis of fractal Image Compression," *IEEE Transactions on Image Processing*, 7 141–154, 1998.

[4] Saupe, Dietmar, "A New View of Fractal Image Compression as Convolution Transform Coding," *IEEE Signal Processing Letters*, 3 193–195, 1996.

[5] Ewing, Gary J.; Woodruff, Christopher J., "Comparison of JPEG and Fractal-Based Image Compression on Target Acquisition by Human Observers," *Optical Engineering*, 35 284–288, 1996.

[6] Jacquin, Arnaud E., "Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformations," *IEEE Transactions on Image Processing*, 1 18–30, 1992.

[7] Kumar, S.; Jain, R. C., "Low-Complexity Fractal-Based Image Compression Technique," *IEEE Transactions on Consumer Electronics*, 43 987–993, 1997.

[8] Jackson, D. J.; Mahmoud, W., "Parallel Pipelined Fractal Image Compression Using Quadtree Recomposition," *The Computer Journal*, 39 1–13, 1996.

[9] Jackson, D. J.; Mahmoud, W.; Gaughan, P. T., "Faster Fractal Image Compression Using Quadtree Recomposition," *Image and Vision Computing*, 15 759–???, 1997.

[10] Mitra, S. K.; Murthy, C. A.; Kundu, M. K., "Technique for Fractal Image Compression Using Genetic Algorithm," *IEEE Transactions on Image Processing*, 7 586–593, 1998.

[11] Palazzari, Paolo; Coli, Moreno; Lulli, Guglielmo, "Massively Parallel Processing Approach to Fractal Image Compression with Near-Optimal Coefficient Quantization," *Journal of Systems Architecture: The EUROMICRO Journal*, 45 765–779, 1999.

[12] Acken, Kevin P.; Irwin, Mary Jane; Owens, Robert M., "A Parallel ASIC Architecture for Efficient Fractal Image Coding," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 19 97–113, 1998.

[13] Barnsley, Michael F.; Sloan, Alan D., "Method and apparatus for compression and decompression of digital image data," *US Patent 5347600*, 23 October 1991.


A VHDL Design for Hardware
Assistance of
Fractal Image Compression
Andrew Erickson